Broadview
www.broadview.com.cn

MANNING

Spark GraphX 实战

[美] Michael S. Malak
Robin East 著
时金魁 黄光远 译

Spark GraphX in Action

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Spark GraphX实战

[美] Michael S. Malak
Robin East 著
时金魁 黄光远 译

Spark GraphX in Action

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书是一本Spark GraphX入门书籍。前5章为基础内容，即使读者对Spark、GraphX、Scala不熟悉，也能快速上手；后5章为图计算进阶，主要是图算法和机器学习算法的相关内容。专门讲图计算的书很少，本书在第2、3、4章介绍了图的基础知识、GraphX基础知识、GraphX内置的图算法。第6章到第10章，主要介绍了GraphX之外的图算法、机器学习、图工具、GraphX监控和优化、GraphX的能力增强等实用技能。第9章和第10章主要介绍性能调优和监控，主要面向生产环境，有不少可以借鉴的技巧。

本书面向对图计算感兴趣的读者，旨在帮助读者掌握Spark GraphX的相关知识及其应用。

版权贸易合同登记号　图字：01-2016-7235

图书在版编目（CIP）数据

Spark GraphX 实战 /（美）迈克尔·S.马拉克（Michael S.Malak），（美）罗宾·伊斯特（Robin East）著；时金魁，黄光远译. —北京：电子工业出版社，2017.4
书名原文：Spark GraphX in Action
ISBN 978-7-121-31043-0

Ⅰ. ①S… Ⅱ. ①迈… ②罗… ③时… ④黄… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字（2017）第044549号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：18.5　　字数：355千字
版　　次：2017年4月第1版
印　　次：2017年4月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

译者序

2016年夏天，明风把我推荐给了电子工业出版社计算机出版分社的张春雨（@永恒的侠少）编辑翻译这本书。当我拿到序言和第6章时，看了一下内容，觉得不是那么复杂难懂。虽然我的英语水平一般，但也可以强迫一下自己，可以查字典，可以找人请教嘛。关键是Spark发展得红红火火，使用GraphX的人还不那么多，图算法的小宇宙被忽略。做一点努力，希望对大家有点用。

以前读过的很多书，几乎每本书的作者都要先感谢自己的老婆和孩子，觉得真矫情。真摊到自己身上，发现没有家人的支持，是不会有时间写书和进行翻译的。所以，我在这里由衷地说一句：感谢老婆。另外，半路拉了黄光远（花名刀剑）壮丁来帮忙翻译第7、8、10章，刀剑对机器学习这部分内容比较熟悉，非常感谢！

这本书在图方面，详细讲解了GraphX的方方面面，还附带介绍了Scala和Spark的各个知识点，算是比较完整。本书涉及Spark的基础知识、Scala的基础知识、图基础知识、GraphX的内置图算法、GraphX之外的常用图算法、基于图的机器学习算法、性能和调优，附录中还列举了GraphX入门的各种工具。总体来讲，这是一本GraphX入门书籍，深入浅出，适合广大对GraphX感兴趣的朋友。读完这本书，要想进一步学习，那就要多看关于图基础、图算法和图论的相关书籍，以及把GraphX的源代码仔细研读几遍。

另外，即使我们把 GraphX 搞得通透，没有实际应用场景也是不行的。GraphX 适用于迭代计算的图算法和关系数据场景，比如，社区发现可用于发现团伙欺诈，PageRank 发现了关系图中有影响力的对象，k-core 用于筛出关键顶点集……其实图算法除适合社交网络外，个人觉得，在互联网金融领域会比较有价值，毕竟反欺诈仅靠离线计算和规则很难及时发现团伙欺诈行为以及个体之间的关系，而互联网金融最核心的事情就是降低风险、提升贷款等业务的审核通过率，以及使通过率和风险接近一个最佳收益平衡点，即当下最赚钱的状态，这是一个技术活。

Spark 转眼已经发展到 2.0 了，发展的速度让我们穿上飞鞋也跟不上。相对于 SQL/Streaming 等模块的快速发展，GraphX 和 Spark Streaming 的发展速度显得有些慢。但是，得益于 Spark 2.0 在性能方面显著的提升，GraphX 还是能沾些雨露的，性能也得到了相应提升。

2015 年年初在淘宝技术部，我曾反复研读 GraphX 的源码，后来社区加了 aggregateMessages() 函数，使性能提升了 15% 左右，再看其整体设计，相当精妙。我觉得 GraphX 是 Spark 中设计最出色的部分，我写了几篇分析文章，网址为 https://github.com/shijinkui/spark_study。在 Spark 的大旗下，相比其他图处理框架，GraphX 有着天然的平台优势，GraphX 在关系数据场景中会有所作为。

我近期在研究 Flink，也看好 Flink 里的 Gelly 和 FlinkML，即实时场景下的图计算和机器学习，我觉得这是一项有前途的技术。

由于本人英语水平有限，也是第一次翻译图书，为方便大家吐槽，在 GitHub 上建立了一个页面：https://github.com/pusuo/graphx_in_action，欢迎大家在这里对本书的翻译吐槽、纠错，发现问题请一定提出来，千万别手软。

时金魁

2016/10/29 于杭州

目录

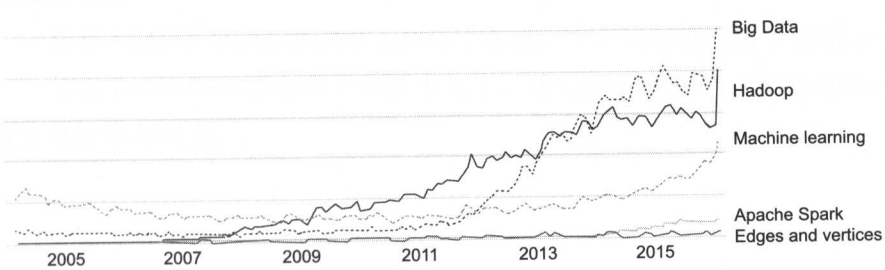
序言

图（Graph）是什么？图是由边和顶点组成的，不是由坐标轴和刻度构成的。在 Spark 中是如何使用图的？这就是本书将要回答的问题。

常常说，“图可以做任何事情”，或者“有很多不同的事情可以用图来实现”。当然了，这两种说法等于什么也没说。所以在本书中我们展示了一些具体的、实际的图应用，以及探讨如何用 Spark GraphX 实现这些图应用。

本书中有许多专业术语：大数据、Hadoop、Spark、图、机器学习、Scala 和函数式编程，这些内容本书都会一一讲解。本书会涉及技术的高级部分，但不会涉及编程能力的每个方面，如 Java 编程。

下图是 Google 在趋势上的统计，展示了这些专业术语在 2016 年之前的受欢迎程度。



注意，通常用 *Spark* 和图作为规范的通用术语，而不是 *Apache Spark* 和 *Edges*

and vertices，趋势上后者已明显被取代。机器学习和图，在计算机科学中有悠久的历史，现在作为主流的大数据技术，在商业领域又引起了新的潮流。如果你在学校学习过这些技术的理论知识，那么现在准备实践一下这些技术吧。

许多我们正在或曾经工作过的公司，已经把 Spark 用在生产环境中了，尽管不一定用了 GraphX。当尝试用 GraphX 做一些图解决方案的原型时，会很方便。如果你已经有了一个 Spark 集群或者决定用云平台上的 Spark 集群（例如 Databricks 或 Amazon），那么无须重新搭建一个新的特定于图计算的集群，并且你可以在 GraphX API 中使用已有的 Spark 技能。现在越来越多的图应用为大家所熟知，从根据 Twitter 数据发掘出恐怖分子网络到根据信用卡交易数据发现欺诈行为，GraphX 已成为一个快速尝试这些图算法场景的易于使用的平台。

本书有两个明确目标：一是全面覆盖 Spark GraphX 的方方面面；二是以读者在前面提到的大数据和图计算方面没有任何专业知识为假定前提。写这本书最大的挑战是要有许多技术储备，特别是 Spark、Scala 和图；了解大量的 GraphX API 以及图的不同用法也是不小的挑战。面对这种情形，本书就需要与其他技术书籍略有不同：首先要花点时间入门，前 5 章主要讲解的就是基础内容；本书还有大量有趣的实例，可以跟着一步步练习。本书中涉及的其他技术，读者需要另做学习，本书将努力做到让读者并不需要有过多背景知识和经验，就可以浅显易懂地了解图所能解决的问题。

致谢

感谢 Manning 出版社许多工作人员对本书出版所做的努力，特别需要感谢三个人，他们的诸多指导使本书可以较好地完成。Marina Michaels，我们的开发编辑，从一开始就指出有些章节有很大的技术问题，这些问题大都是 Spark 和图计算的新问题。Michael Roberts，我们的技术开发编辑，在本书的制作过程中，与 Marina 一样，他给出了大量的建议。Antonio Magnaghi，我们的技术校对，不但对本书的示例代码进行了严谨的核对，还编辑校对了本书的文本内容。

同时也感谢本书草稿时期给出诸多有价值建议的读者，他们是 Andy Petrella，Brent Foust，Charles Feduke，Gaurav Bhardwaj，Jason Kolter，Justin Fister，Michael Bright，Paul-Michael Sorhaindo，Rodrigo Abreu，Romi Kuntsman，Sumit Pal，Vincent Liard。

作者 Michael Malak 感谢妻子和孩子在这几个月写作期间给予的耐心支持。

作者 Robin East 感谢妻子和两个儿子，他们容忍和支持了作者长时间的写作以及在楼上偶尔消失。

关于本书

通过学习本书，希望能降低难懂的图学习门槛，了解如何在市场份额最大的分布式计算框架 Apache Spark 中开发图应用。

本书的读者对象

我们假定本书的读者并不熟悉 Spark、Scala 和图相关的知识，本书会快速学习前面提到的这些知识，会特别侧重于 Scala。在第 3 章有 Scala 的简要介绍，全书中只要出现新的 Scala 知识点都会有 Scala 小贴士做详细介绍（完整的列表见附录 D）。实际上，本书通过第 3 章、Scala 小贴士和附录对 Scala 做了较全面的介绍。

另外，虽然在大学的图论课程中数学证明很常见，但本书完全不做数学证明。本书的目标是图算法和图应用，有时会应需介绍图相关的术语。

本书使用的是 Spark/GraphX 1.6 版本。

我们假定读者在 Java 语言编程方面有一些经验，而在图方面要求不多，但通过书中插图能自然地知道这些图应用。

本书的内容组织

本书分为 3 个部分。第 1 部分有 3 章，主要介绍使用 Spark GraphX 的准备知识。

第 2 部分有 4 章，主要介绍如何使用 GraphX。第 3 部分有 3 章，主要介绍 GraphX 的进阶知识。

也可以将本书分为两部分，前 5 章为准备知识和 GraphX 的基本 API，后 5 章为 GraphX 应用。

下面是各章的内容提要。

- 第 1 章介绍了什么是大数据、Spark 和图，Spark GraphX 如何处理数据流。第 1 章是一本迷你书，篇幅不长但内容涉及面较广。
- 第 2 章简单示范了如何使用 GraphX，无须具有 GraphX 经验。
- 第 3 章介绍了 Spark、Scala 和图的基础知识。
- 第 4 章介绍了 Spark GraphX 的基础操作，以及如何使用 GraphX 的两个主要算法：Map/Reduce 和 Pregel。
- 第 5 章演示了如何使用 GraphX 的诸多内置算法。
- 第 6 章介绍了 GraphX API 之外的内容，即 20 世纪中期经典的图算法，并用 GraphX 实现了这些算法。
- 第 7 章重点讲机器学习。机器学习的内容本身就够讲一本书的，这里没有讲解太多机器学习的基础知识和经验，而是直接介绍监督学习、无监督学习和半监督学习的高级实例。
- 第 8 章展示了 GraphX 如何完成一些自定义操作，有可能会构建一个图处理库：读 RDF 文件、图的合并、图查找和计算全局聚类系数。
- 第 9 章介绍了如何监控性能以及查看正在执行的 GraphX 应用程序，如何利用缓存、checkpointing 和序列化调优做性能调优。
- 第 10 章介绍了在 GraphX 中如何使用 Scala 之外的语言（强烈建议不要这么做），以及如何使用一些工具来补充 GraphX 的不足。展示了用 GraphX 在 Apache Zeppelin 的交互式命令行 notebook 上对图进行可视化。第三方的工具 Spark JobServer 可以让 GraphX 从单纯的批处理系统转变成一个在线图数据库。最后，介绍了 Github 上的一个项目——GraphFrames（GraphX 的开发者开发的），它用 Spark SQL DataFrames 而非 RDD 提供了一种方便和高性能的图查询方式。

另外，本书包含 4 个附录。附录 A 介绍了 Spark 的安装方法，附录 B 简要介绍

了 Gephi 可视化软件，附录 C 包含关于 GraphX 的在线资料以及如何跟上社区最新进展，附录 D 中列出了本书中的 Scala 小贴士。

如果你在 Spark、Scala 或图方面是新手，通过前 5 章的阅读，能力可以得到提升。然后，就可以选择性地阅读后面 5 章的内容了。

如果你对 Spark、Scala 和图的知识已经比较精通，但对 GraphX 还不熟悉，那么可以跳过前 3 章甚至前 5 章的内容直接阅读后面的内容。

关于本书中的代码

本书中的源代码可以在博文视点官方网站上下载，地址为 https:// www.broadview.com.cn。

这本书中的大部分代码是可以在交互式的 Spark Shell 中执行的。从技术上来讲，Scala 扩展是一个误称，因为这些文件不能用 scalac 编译器进行编译。

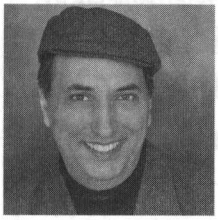
要想让这些实例代码方便地编译和执行，可以用 Maven 的 pom.xml 或 SBT 的 .sbt 文件完成。

本书中的源代码实例，有带编号的列表，也有普通的文本，这两类源代码都用了等宽字体以便与普通的正文区分开来。

一般情况下，源码都会被格式化，我们添加了换行符，也修改了缩进以便适应本书的页面大小，甚至在代码清单中添加了续行符（➥）。另外，如果正文中对代码有解释，源码中的注释会被删掉。代码注释会附带许多列表，以突出重要的概念。

本书的示例代码也可以从如上的出版社网站上下载。

关于作者



Michael Malak 一直从事软件开发工作，自 2013 年年初以来他一直用 Spark 为财富 200 强的公司做开发工作，经常进行演示和分享，特别是在科罗拉多州他住的丹佛 / 博尔德地区。他的个人技术博客的地址是 http://technicaltidbit.com。

Robin East 在一些大型企业曾担任过 15 年以上的顾问，在金融、政府、医疗保健和公共事业领域提供大数据和智能解决方案。他是 Worldpay 的数据科学家，帮助公司实现把数据用于核心业务上。可以在这里看到他在 Spark、GraphX 和机器学习方面的作品：https://mlspeed.wordpress.com。

配套服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），即可享受以下服务。

下载资源：本书所提供的示例代码及资源文件均可在“下载资源”处下载。

提交勘误：你对书中内容的修改意见可在“提交勘误”处提交，若被采纳，将获赠博文视点社区积分（在购买电子书时，积分可用来抵扣相应金额）。

与我们交流：在页面下方“读者评论”处留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31043

二维码：

关于封面插图

本书封面插图的标题为“来自克罗地亚达尔马提亚的希贝尼克的人”。插图是由Nikola Arsenovic取自19世纪中叶克罗地亚传统服装专辑画册，2003年由克罗地亚斯普利特的民族博物馆出版。插图是从斯普利特民族博物馆的一个热心的图书管理员那里得到的，博物馆坐落于中世纪的罗马镇中心，大约公元304年皇帝戴克里先的宫殿所在的废墟。这本书中有来自克罗地亚不同地区的精细彩色的人物插图，描绘了当地特色服装和日常生活。

希贝尼克是克罗地亚的一个历史名城，位于达尔马提亚中部，克尔卡河在那里流入亚得里亚海。希贝尼克是克宁县的政治、教育、工业和旅游中心，是达尔马提亚历史地区的第三大城市。它是亚得里亚海岸线上最古老的克罗地亚城镇。

服装风格和生活方式在过去200年中发生了很大变化，当地特有的多样性逐渐消失。现在很难分辨不同大陆的居民，更不用说相距只有几英里的不同的小村庄或城镇。也许我们已经将文化多样性变为更多样化的个人生活，特别是为了适应更多样化和快节奏的技术生活。Manning出版社通过在封面上呈现两个世纪之前的丰富的区域文化生活来庆祝今天计算机科技所体现的创造性和主动性，并将读者带回到来自旧书和收藏品中的插图所显示的生活中。

第1部分 Spark和图

图是由顶点和边组成的，并非代数（Algebra）中的图概念。图看起来非常强大，使用起来有一些窍门。类似这种“图可以做任何事情”的说法是毫无意义的。废话少说，第 1 章，我们从现实中发现图的分类开始介绍。在第 3 章会介绍图的相关术语。

Apache Spark 是一个以高性能著称的分布式计算框架。GraphX 是 Spark 在图计算方面的应用，第 1 章描述了 GraphX 是如何满足数据处理工作流的。第 2 章会介绍 Google 发起的 PageRank 算法。

第 3 章是一个快速教程，介绍本书要求掌握的三种基础技术：Spark、Scala 和图。

1 两项重要的技术：Spark和图

本章要点

- Spark 为何能成为领先的大数据处理系统
- 是什么让图成为模型化连接数据的唯一方法
- GraphX 如何让 Spark 成为图计算和分析的首要平台

众所周知，我们正在产生比以往更多的数据。然而并不是只有数据本身重要，数据之间的联系同样重要。从这些有关联的数据集中抽取出有用的信息，可以深入了解很多领域，如检测欺诈、收集生物信息学数据和进行 Web 网页排名等。

图提供了一种强大的方式来表示和利用数据间的连接。图表示由顶点和顶点连接的边构成的数据点网络。图可以用于各种领域，如计算机视觉、自然语言处理和推荐系统等。

用这样一个连接的数据（connected data）的表示，以及一些工具和技术，就可以挖掘来自网络的信息内容。在本书中涉及的众多工具中，你会发现 PageRank 可以用来发现网络中最有影响力的成员，隐含狄利克雷分配（LDA，Latent Dirichlet Allocation）用于话题模型（topic modeling），聚类系数用来发现高度联通的社区等。

用于分析数据的传统工具如关系型数据库，并不适合解决这类问题。面向表（table-oriented）的框架，如用 SQL 来表示一个如追踪链的经典图概念就显得很笨重。此外，当被分析的数据量增加时，传统的数据处理方法不能扩容。

一个现成的解决办法是用图处理系统。这样的系统提供了数据模型和编程接口，以便更自然地查询和分析图。图处理系统提供了多种方法从原始数据源构建图，应用处理函数到图中，以便挖掘其中的信息内容。

Apache Spark 是取代 Hadoop 的大数据处理的可选方案，Spark 这个开源数据处理平台引领着大数据时代。因易于伸缩数百节点的集群，Spark 的内存数据处理能力胜过 Hadoop 很多倍。

GraphX 是一个在 Spark 之上的图处理层，带来了因为图数据太大单机无法处理的强大的图大数据处理能力。很久之前人们就开始在图处理方面使用 Spark，包括 Bagel 这类预处理模块，现在有了标准的图计算模块 GraphX，它提供了一些常用的图算法库。

选择使用 Spark GraphX 的理由，包含以下几种。

- 你已经有了 Spark 的数据处理流水线，想混合使用图处理。
- 你好奇 Spark 或 GraphX 的强大能力。
- 图数据计算对你来说很重要。
- 图数据太大，单机无法处理。
- 要么你不需要多个应用访问相同数据仓库，要么你计划增加一个 REST 服务器到 Spark，如增加 Ooyala 的 Spark Job Server。
- 要么你不需要数据库类型的事务，要么你计划用 Neo4j 或 Titan 这样的图数据库结合 GraphX 一起使用。
- 你已经有了一个可用的 Spark 集群。
- 你喜欢 Scala 语言简明强大的表达能力。

1.1 Spark：超越Hadoop MapReduce

在这一节中，我们讨论 Spark 和图的关系。大数据对一些数据科学团队来说是主要的挑战，因为在要求的可扩展性方面单机没有能力和容量来运行大规模数据处理。此外，即使专为大数据设计的系统，如 Hadoop，由于一些数据的属性问题也很

难有效地处理图数据，我们将在本章的其他部分看到这方面的内容。

Apache Spark 与 Hadoop 类似，数据分布式存储在服务器的集群或者是“节点”上。不同的是，Spark 将数据保存在内存（RAM）中，Hadoop 把数据保存在磁盘（机械硬盘或者 SSD 固态硬盘）中，如图 1.1 所示。

定义：在图和集群计算方面，“节点”这个词有两种截然不同的意思。图数据由顶点和边组成，在这里“节点”与顶点的意思相近。在集群计算方面，组成集群的物理机器也被称为“节点”。为避免混淆，我们称图的节点为顶点，这也是 Spark 中的专有名词。而本书中的“节点”这个词我们严格定义为集群中的单个物理计算节点。

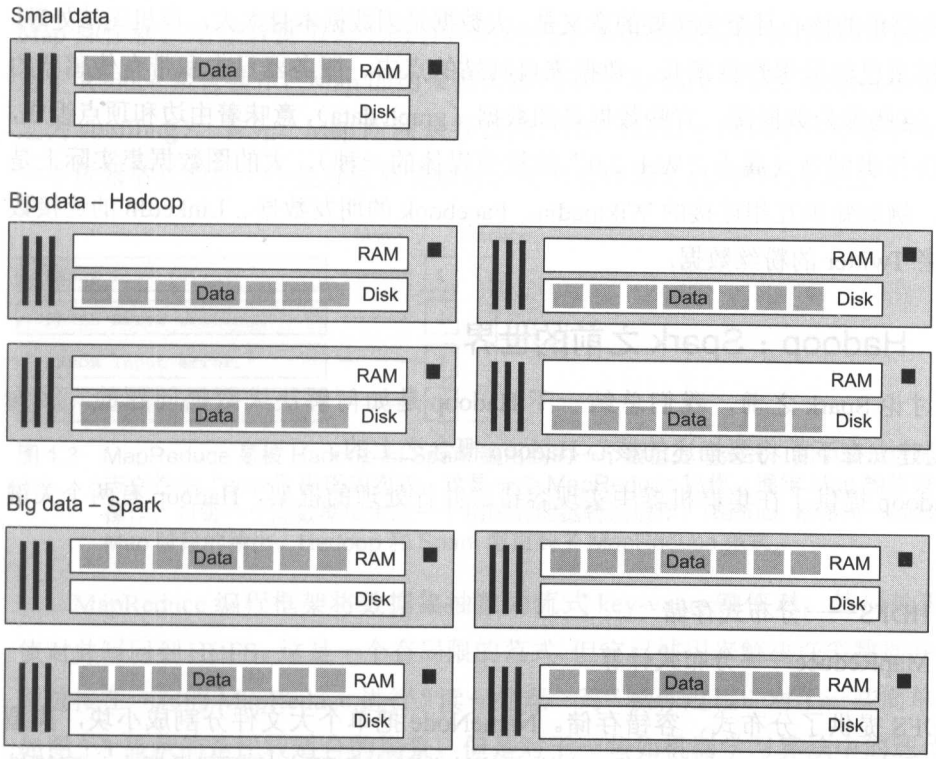


图 1.1 大数据因为数据量大单机无法处理。Hadoop 和 Spark 都是把数据分布在集群节点上的分布式框架中。Spark 把分布式数据集存放在内存中，所以比 Hadoop 把数据存放在磁盘中处理速度要快很多。

除了将要计算的数据保存的位置不同（内存和磁盘），Spark 的 API 比 Hadoop 的 Map/Reduce API 更容易使用。Spark 使用简洁且表达力较好的 Scala 作为原生编

程语言，写 Hadoop Map/Reduce 的 Java 代码行数与写 Spark 的 Scala 的代码行的数量比一般是 10:1。

虽然本书主要使用 Scala，但是你对 Scala 不熟悉也不用担心，我们在第 3 章提供了快速入门，包括怪异、晦涩和简练的 Scala 语法。进一步熟悉 Java、C++、C#、Python 等至少一门编程语言是必要的。

1.1.1 模糊的大数据定义

现在的“大数据”概念已经被很大程度地夸大了。大数据的概念可以追溯到 Google 在 2003 年发表的 Google 文件系统的论文和 2004 年发表的 Map/Reduce 论文。

大数据这个术语有多种不同的定义，并且有些定义已经失去了大数据所应有的意义。但是简单的核心且至关重要的意义是：大数据是因数据本身太大，单机无法处理。

数据量已经呈爆炸性增长。数据来自网站的点击、服务器日志和带有传感器的硬件等，这些称为数据源。有些数据是图数据（graph data），意味着由边和顶点组成，如一些协作类网站（属于“Web 2.0”的社交媒体的一种）。大的图数据集实际上是众包的，例如知识互相连接的 Wikipedia、Facebook 的朋友数据、LinkedIn 的连接数据，或者 Twitter 的粉丝数据。

1.1.2 Hadoop：Spark 之前的世界

在讨论 Spark 之前，我们总结一下 Hadoop 是如何解决大数据问题的，因为 Spark 是建立在下面将要描述的核心 Hadoop 概念之上的。

Hadoop 提供了在集群机器中实现容错、并行处理的框架。Hadoop 有两个关键能力：

- HDFS——分布式存储
- MapReduce——分布式计算

HDFS 提供了分布式、容错存储。NameNode 把单个大文件分割成小块，典型的块大小是 64MB 或 128MB。这些小块文件被分散在集群中的不同机器上。容错性是将每个文件的小块复制到一定数量的机器节点上(默认复制到 3 个不同节点，图 1.2 中为了表示方便，将复制数设置为 2)。假如一个机器节点失效，致使这个机器上的所有文件块不可用，但其他机器节点可以提供缺失的文件块。这是 Hadoop 架构的关键理念：机器出故障是正常运作的一部分。

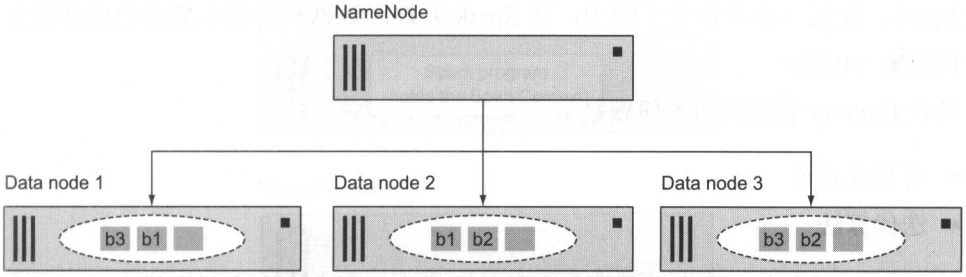


图 1.2 三个分布式数据块通过 Hadoop 分布式文件系统（HDFS）保持两个副本。

MapReduce 是提供并行和分布式计算的 Hadoop 并行处理框架，如图 1.3 所示。用 MapReduce 框架，程序员写一个封装有 map 和 reduce 函数的独立代码片段来处理 HDFS 上的数据集。为取到数据位置，代码打包（jar 格式）分发到数据节点，Map 操作就在这些数据节点上执行，这避免了集群的数据传输导致消耗网络带宽。对于 Reduce 聚合操作，Map 的结果被传输到多个 Reduce 节点上做 reduce 操作（称之为 shuffling）。首先，Map 阶段是并行操作的，Hadoop 提供了一个弹性机制，当一个机器节点或者一个处理过程失败时，计算会在其他机器节点上重启。

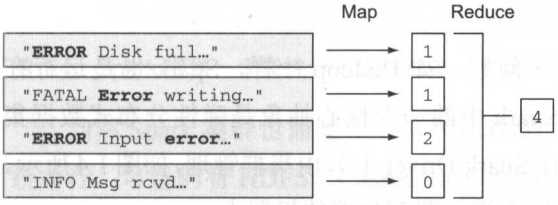


图 1.3 MapReduce 是被 Hadoop 和 Spark 都用到的一个数据处理范式。图中表示计算服务器日志文件中“error”出现的次数，这是一个 MapReduce 操作。通常 Map 操作是一对一的操作，对每一个源数据项生成一个相应的数据转换操作。Reduce 是多对一的操作，聚合 Map 阶段的输出。Hadoop 和 Spark 都用到了 MapReduce 范式。

MapReduce 编程框架将数据集抽象为流式 key-value 键值对，然后处理这些键值对并写回到 HDFS。这是一个有局限的范式，但它已被用来解决许多数据并行问题，用链接在一起的 MapReduce 进行“读－处理－写”操作过程。对于一些简单的任务，如图 1.3 显示的是比较适合的场景。但是对于一些如机器学习算法中的迭代计算算法，用这种 MapReduce 范式就很痛苦，这也是选择使用 Spark 的原因。

1.1.3 Spark：内存中的 MapReduce 处理

在这一小节，我们来看另一个可选的分布式处理系统，构建在 Hadoop 基础之

上的 Spark。在这一小节你会了解到，在 Spark 处理图数据时扮演重要角色的弹性分布式数据集（RDD）。

导致 Hadoop 衰落的两类问题是：

- 交互式查询
- 迭代算法

Hadoop 很适合在一个大的数据集上做单次查询，而在许多实际场景中，一旦有了一个想要的答案，我们就想再问数据一个问题，这就是交互式查询。使用 Hadoop 的话，就意味着要等待重新从磁盘中加载数据，再次处理数据。我们不得不执行一组相同的计算作为随后分析的前提，这不符合常理。

迭代算法已经被广泛应用于机器学习任务，如随机梯度下降算法，以及之后会看到的 PageRank 这类图计算算法。迭代算法是在一个数据集上一遍又一遍地做一组计算，直到满足一个标准（循环结束条件）才结束迭代。 在 Hadoop 中实现这种算法，一般需要一系列加载数据的 MapReduce 任务，这些 MapReduce 任务要在每一个迭代过程中重复运行。对于非常大的数据集，每个迭代过程要花费 100 秒或 1000 秒，整个迭代过程非常耗时。

下面你会看到 Spark 如何解决这些问题。如 Hadoop 一样，Spark 也是运行在一个常见的硬件配置的机器集群上。Spark 中的一个核心抽象是弹性分布式数据集（RDD）。RDD 是由 Spark 应用创建的(在 Spark Driver 上)，由集群管理，如图 1.4 所示。

组成 RDD 分布式数据集的数据分区会被加载到集群的机器上。

基于内存的数据处理

Spark 执行的大部分操作都是在随机访问内存中（RAM）进行。Spark 是基于内存的，而 Hadoop Map/Reduce 是顺序处理数据，所以 Spark 比 Hadoop 更适合处理随机访问的图数据。

Spark 的关键好处在于交互式查询和迭代处理过程中在内存中缓存 RDD。缓存起来的 RDD 可以避免每次重新处理父 RDD 链，而只需要直接返回父 RDD 计算后的缓存结果。

自然的，这意味着要用到 Spark 的基于内存的计算处理特性，要求集群中的机器内存要足够大。要是可用内存不够，那么 Spark 就会优雅地溢出数据到磁盘，以保证 Spark 能继续运行。

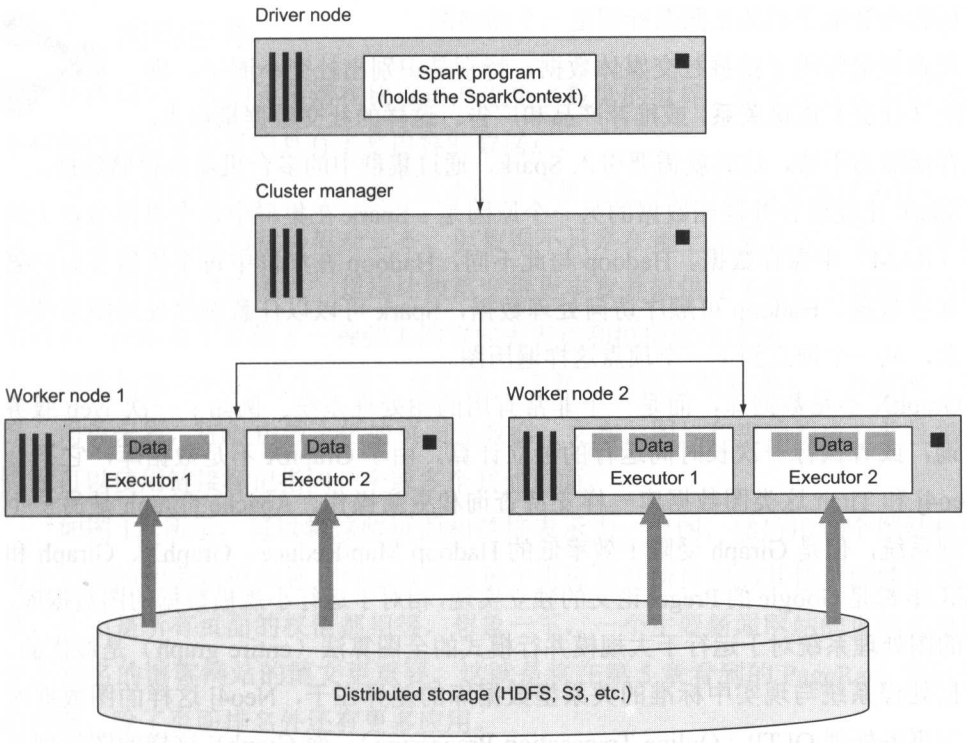


图 1.4 Spark 提供一个弹性分布式数据集，可以认为它是一个分布式的常驻内存的数组。

当然 Spark 集群也需要一个持久化存储数据的地方，而且还要是分布式存储系统才行，可选的有 HDFS、Cassandra 和亚马逊的 S3。

1.2 图：挖掘关系中的含义

图可以用来表示自然发生的连接数据，如：

- 社交网络
- 移动网络
- 互联网 Web 页面

图曾受限于学术界和研究界几十年，但在过去几年中已经被多个组织采用，从硅谷社交媒体公司到政府情报机构，用于在数据中寻找和使用关系模型。图现在甚至被纳入流行词汇，Facebook 推出“图搜索”，政府情报机构公开呼吁需要“连接点”，以及一款称为 Kevin Bacon 六度的古老的互联网模仿游戏。甚至在社交媒体上的“分

享”图标和智能手机的拍照图标都是一个缩略图。

现在图最常用于挖掘社交媒体数据，特别是识别出社交小圈子、推荐新的（社交）连接关系，或推荐产品和广告。这样的社交数据量很大，单机存储能力不够，这时就需要引入 Spark，通过集群中的多台机器来存储数据。

Spark 比较适合处理图数据的另一个原因是：Spark 在集群中每个机器节点上的内存（RAM）中保存数据。Hadoop 与此不同，Hadoop 在集群中每个机器节点的磁盘中保存数据。Hadoop 可顺序访问处理数据，Spark 可以以任意顺序处理图系统中的数据，从一个顶点到下一个顶点这样遍历图。

GraphX 不是数据库，而是一个非常有用的图处理系统。例如：一次 Web 服务的查询，或者执行一次长时间运行的独立计算。由于 GraphX 不是数据库，它不会像 Neo4j 和 Titan 这类图数据库一样支持查询和更新操作。Apache Giraph 是另一个图处理系统，但是 Giraph 受限于效率低的 Hadoop Map/Reduce。GraphX、Giraph 和 GraphLab 都是 Google 的 Pregel 论文的独立实现，相对于运行小规模数据的图数据库，这样的图处理系统对于运行于大规模并行模式的全图算法（entire graph）是最优的。图数据处理系统与现实中标准的关系型数据库的差异在于，Neo4j 这样的图数据库是在线事务处理 OLTP（Online Transaction Processing），而 GraphX 这样的图处理系统是在线分析处理 OLAP（Online Analytical Processing）。

图可以存储各种类型的数据：地理空间位置、社交媒体、论文引用网络和网页链接等。一个小的社交媒体网络数据图如图 1.5 所示：Ann、Bill、Charles、Diane 和 Went to gym this morning 是顶点，而 is-friends-with、wrote-status 和 likes-status 是边。

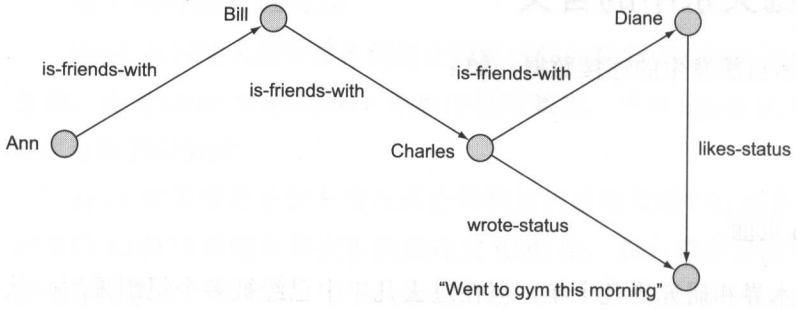


图 1.5 如果 Charles 与他的朋友的朋友分享他的状态，要是只用表或者数组来计算谁能看到 Charles 分享的状态，就显得很笨重、不灵活。

1.2.1 图的应用

众所周知，我们正生活在比以前产生更多数据的世界。我们正在从不断增加的多种数据源收集更多的具有丰富内容的数据点。

为了利用这种形势，一些或大或小的组织将数据分析和数据挖掘视为核心业务，这一转变有时被称为*数据驱动业务*。但数据不只是在量上变得更大，更是在数据的连接上。所谓的“连通性”是指让数据变得更丰富并提供更多机会去理解我们周围的世界。图给我们提供了一种强大的方式去表达和挖掘这些连接。

数据连接的形式是什么呢？我们从一个熟知的连接数据集——万维网开始探究。简单来看，Web 由数十亿的页面元数据、文本、图片和视频构成，并且每一个页面可以使用链接标记指向一个或多个其他页面。

如图 1.6 所示，可以将这些页面和链接表示为一个图，然后用这个图结构提供每个页面的相关权值信息。可以可视化每个页面及其所指向的其他页面的投票。当然，并不是所有页面的权值都相等，想象一下，一个主要新闻网站的页面当然比一个不知名的博客网站的博文更重要。这就是将在第 5 章看到的 PageRank 算法解决的问题，除了页面排名外还有更多应用。

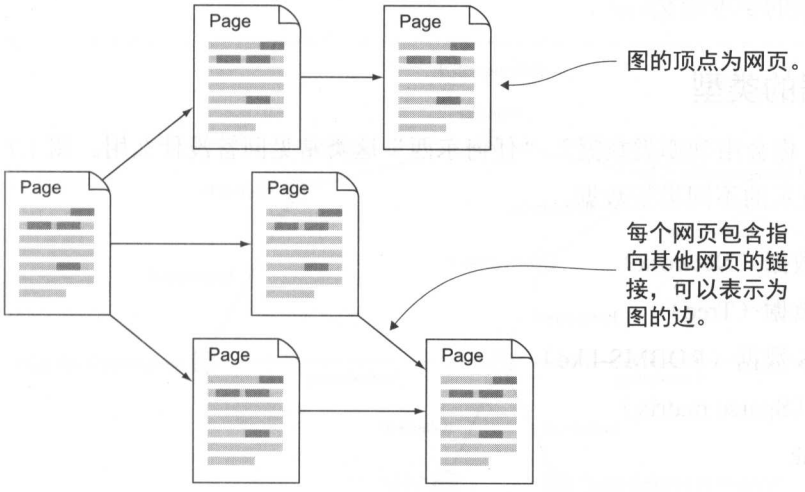


图 1.6 网页之间的链接可以被认为是一个图。这个图结构提供了关于每一个网页的相关来源、排名信息。

到目前为止，我们看到的图已经简单刻画了页面之间的链接，页面之间可能有链接，也可能没有链接。如果我们有更多关于连接的信息，就可以使图更丰富。一

个典型的例子是给信息评分。当我们给 Netflix 上的一个电影打了五星，这不仅创建了我们和电影之间的联系，也将值赋给该连接。

给电影打分不是应用到图连接的唯一有价值的场景。还有一些丰富图中连接的例子，如：资产中的金融欺诈分析、城市之间的距离和通过移动电话基站网络分析交通运输。

即使数据点之间的连接没有一个可衡量的数值，仍然可以在图中捕获有价值的信息。以社会化媒体网站为例，每个人的 profile 概况会存储一个人去学校的细节，这表示人与学校之间的联系。如果我们获取到其他信息，如什么时间在校，我们就有更多的信息表示在图中。现在当想向用户展示推荐的朋友，我们可以确保如果他是 83 班的，我们不会显示 96 班的朋友推荐。

当然图在社交网络出现之前就存在了，其他图应用有：

- 在地图应用中找到最短路径。
- 基于与其他人的相似度图，推荐产品、服务、人际关系或媒体。
- 将一个相互关联的主题转换成一个有层次的组织方案（类似计算机文件系统文件夹、一个课程大纲等）。
- 确定最权威的学术论文。

1.2.2 图数据的类型

在一个图中，你会用到哪类数据？“任何东西”这类常见回答没什么用。图 1.7 所示为可以被图表示的不同类型数据。

- 网络结构数据（Network）
- 树状结构数据（Tree）
- 类 RDBMS 数据（RDBMS-like）
- 稀疏矩阵（Sparse matrix）
- Kitchen sink

网络结构数据图可以是一个路径网络，或者是一个社交网络，或者是一个计算机网络。树状图是无环图。任何 RDBMS 都可以被转换成图的格式，如图 1.7 所示的一个雇员信息数据库被转换成了图。然而要有一些图算法这样做才有价值，如 PageRank 用于社区发现，或最小生成树算法用于网络规划。

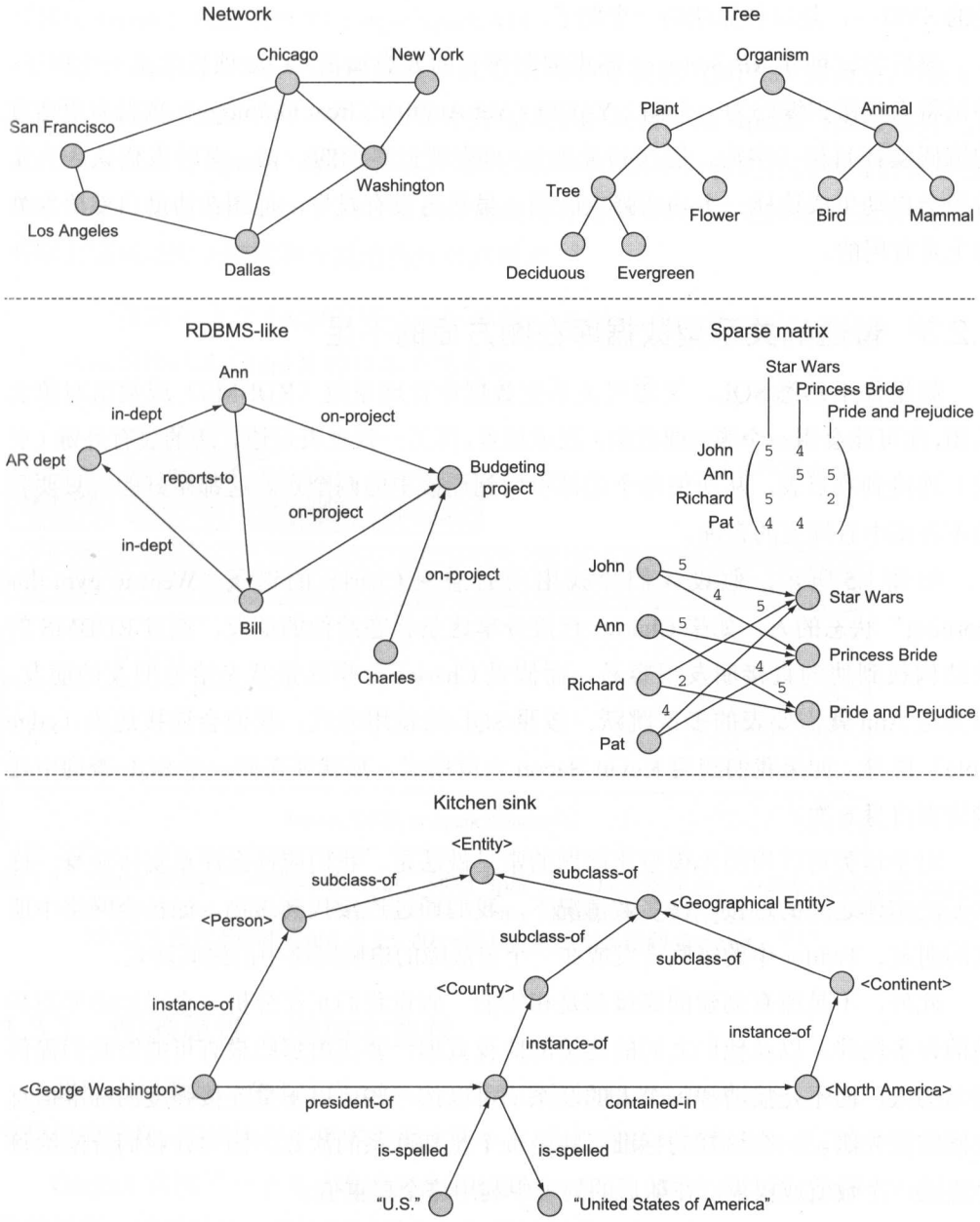


图 1.7 可以用图表示的不同数据类型。

如第 3 章讨论的，每一个图都有一个关联的邻接矩阵。这个概念有一个重要的含义：图只是一个可供替代的数据结构，没什么神奇的。一些要处理庞大矩阵数据的算法可以利用图的更高压缩比的表示方法，特别是“稀疏矩阵”。第 7 章中所讨

论的 SVD++，是这种算法的一个例子。

最后尝试创建 *Kitchen sink* 图来编码所有的人类知识。Cyc 项目就是一个例子，尝试将人类常识编码为一个图。YAGO（Yet Another Great Ontology）项目有更适度的编码实体目标（字典、层次和关系），可表现世界上的一切。有时人们认为人工智能会自动生成这样一个功能强大的图。虽然这没有发生，但图在协助自然语言处理上是有用的。

1.2.3 普通的关系型数据库在图方面的不足

如果你不熟悉 SQL，又想用关系型数据库管理系统（RDBMS）或数组对象表示图，你可能会用一个表（或数组）表示顶点，用另一张表表示边。边的表有外键（引用）连接到顶点表，从而使每个边都引用到连接中的两个点。这都是好的，只要我们不在图中有深度的查询。

如图 1.5 所示，假设我们要找出所有看到 Charles 的这条“Went to gym this morning”状态的人，如果 Charles 直接分享这条状态给他的朋友，通过 RDBMS 的表结构找到他的直接朋友很容易。而假设 Charles 分享这条状态给他朋友的朋友，要到达 Ann 就需要表的多次跳跃。按照 SQL 的使用方式，我们会连接边表（edge table）自身。如果我们想用 Kevin Bacon 六度模式，那就要在同一个 SQL 查询中连接边表自身 6 次。

对于这类可以用图来模型化问题的常见做法是，我们应该像注重实体自身一样来专注实体之间的连接。在多数情况下，我们通过连接找到事物，如社会网络中朋友的朋友、Twitter 中的级联转发或在一个出故障的电脑网络中的级联转发。

此外，不是所有创建的连接都是相等的。假设我们正在分析一个已知的罪犯和他的许多同伴，以及他们之间的连接的监视数据，要找出那些最有可能给我们提供信息的人，而不是搞清和每个人的联系；可以按一些可以衡量连接强度的标准信息来确定优先级。一个这样的标准可能是一个星期联系的次数。图允许我们分配给每个连接一个数值或权重，在随后的处理中利用这个权重值。

1.3 把快如闪电的图处理放到一起：Spark GraphX

GraphX 是 Spark 中的顶层模块，提供了由 Spark RDD 组成的图数据结构，并提供了 API 来操作这些图数据结构。GraphX 随着标准的 Spark 版本一起发布，你可

以使用 GraphX 规范 API 和正规的 Spark API。

Spark 起源于 2011 年加利福尼亚大学伯克利分校的 AMPLab 实验室，2014 年成为 Apache 的顶级项目。并不是所有的 AMPLab 项目都是 Apache Spark 官方的正式产品。要运行 Spark，需要图 1.8 下部所示的两个层：分布式存储和集群管理器。在本书中，我们选择 HDFS 为分布式存储而没有集群管理器，在单机上运行 Spark；实际上这只是用于测试和开发的伪分布式模式。

注意： 由于 GraphX 是 Spark 的核心基础组件，所以 Spark Core 包、基础组件以及 GraphX 的版本是同步的。

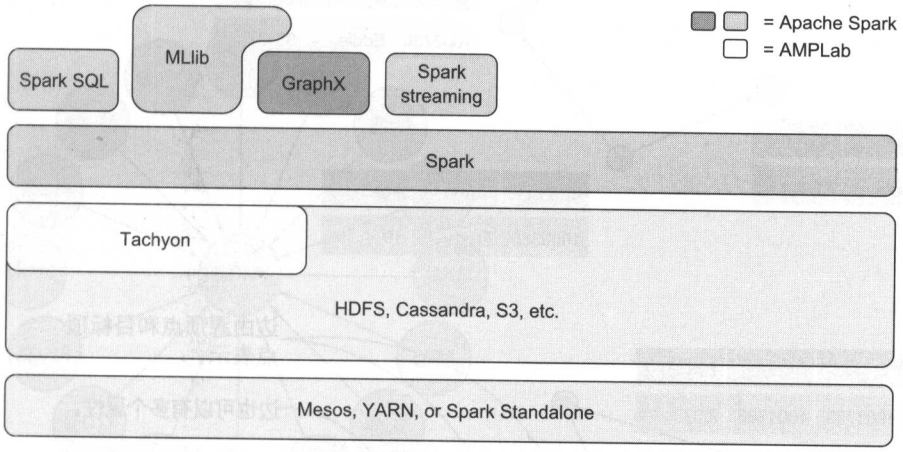


图 1.8 Spark 技术栈。一类如 GraphX，伴随 Spark 一起成长的组件，一类如 HDFS 和 YARN，来自 Apache Hadoop。最后一类 Tachyon，来自加州大学伯克利分校 AMPLab 实验室。MLlib 大多在 Spark Core 之上，但一些 ML 算法也会用到 GraphX。[译注 1]

1.3.1 图的属性：增加丰富性

我们已经看到，在现实世界中，在简单的顶点和边之间的连接之外，还有一些有价值的信息。图因为有了更多数据才更丰富，我们需要一种方式来表示这个丰富性。

GraphX 实现了一个属性图的概念。如图 1.9 所示，顶点和边可以有与它们相关联的任意一组属性集。属性可以是人的年龄这种简单的数据，或者是文件、图像和视频这类复杂数据。[译注 2]

译注 1 Tachyon 已经正式更名为 Alluxio，更多信息可参见 http://www.alluxio.org，当前版本为 1.1.0。

译注 2 Spark 官方的定义是：The property graph is a directed multigraph with user defined objects attached to each vertex and edge。

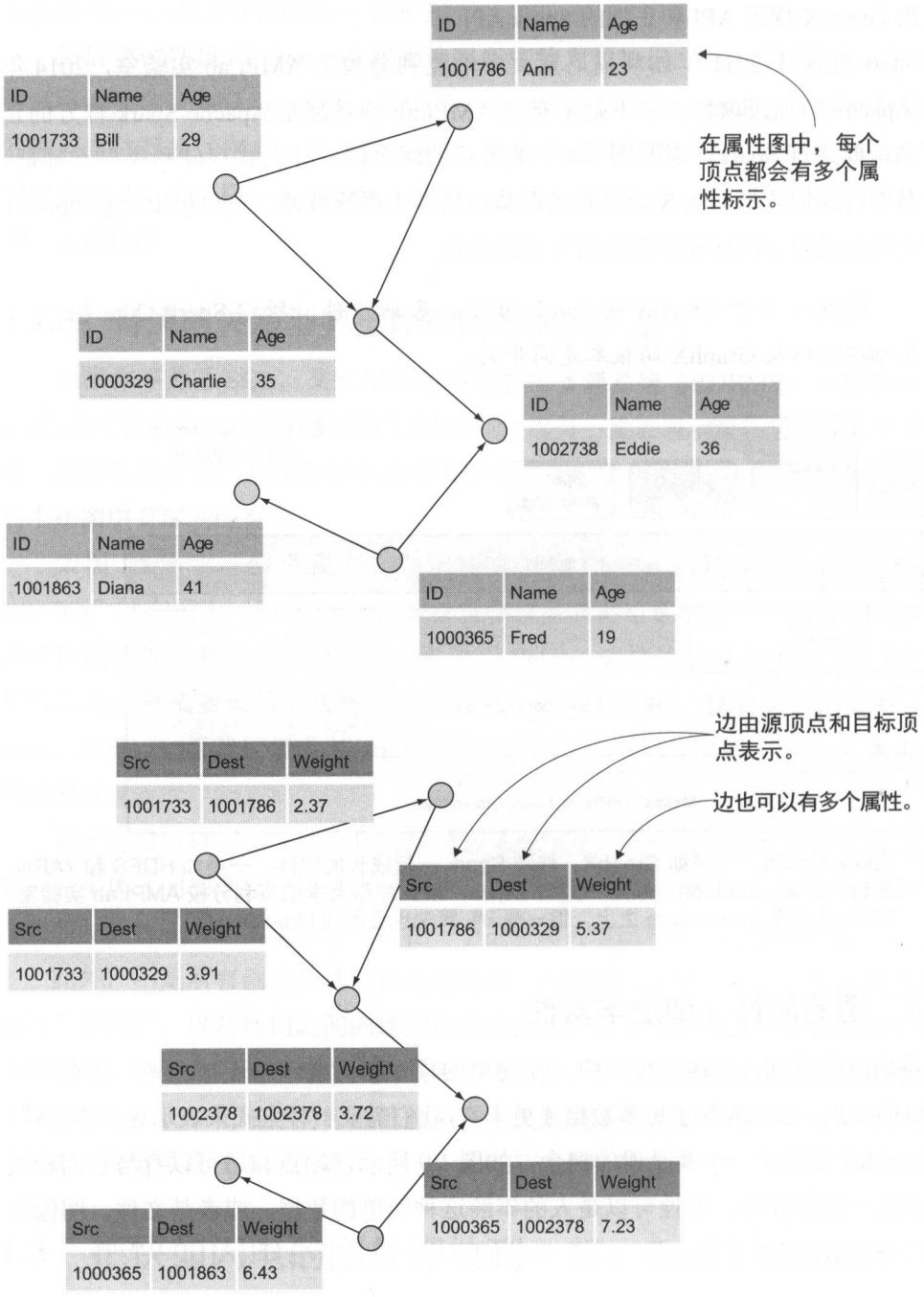


图 1.9 图上的边和顶点包含一些额外属性。

GraphX 用边 RDD 和顶点 RDD 这两个 RDD 来表示一个图，以这种方式表示的图可以让 GraphX 解决一个处理大图的主要问题：分区。

1.3.2 图的分区：当图变为大数据集时

遇到单机的物理内存放不下图的情况时，Spark 就可以把这些图划分到集群中的多台机器上。切分图的最佳方式是什么呢？

多年来常用的简单切分图的方式是，分配不同的顶点到集群中的不同机器节点上。但这导致了计算瓶颈，因为实际中有一些具有非常高的度的顶点，如图 1.10 所示。真实世界中的图的顶点的度趋向于遵循幂律。

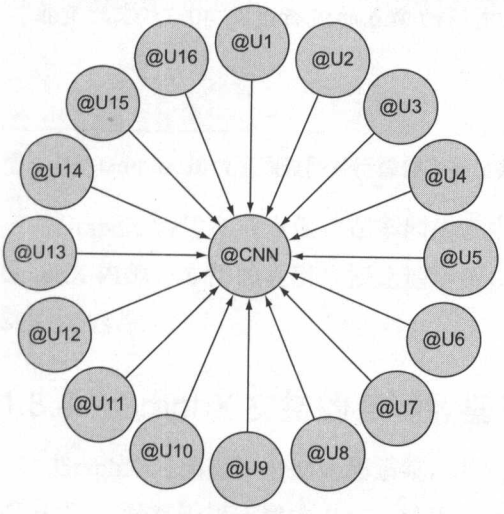


图 1.10 一个有“高度”的图。

定义：度在图中有两个含义。前面提到 Kevin Bacon 的六度的意思是跳跃查询的次数或者边的条数，例如：从一个演员到另一个演员（成为一个边），这个边意味着两个演员碰到了一起。而图中的顶点的度的意思完全不同，顶点的度的含义是，从一个特定顶点连接出去的边数（出度）和连接进来的边数（入度）之和。我们不会再提 Kevin Bacon 的六度，而是用“跳跃”来解释六度里的度，用顶点与邻近的边数来表示度。

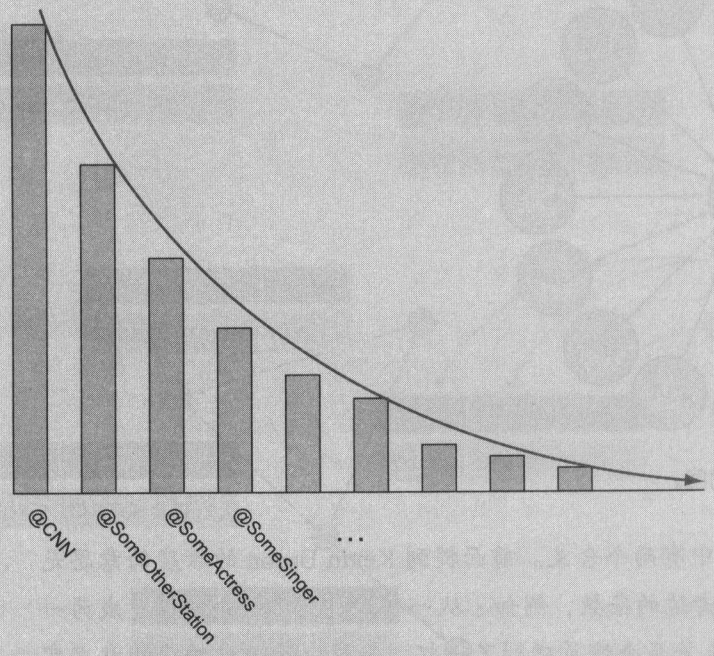
对一个图通过顶点进行切分称为*边切*，意思是把边切开。而图处理系统采用*顶点切分*，也就是说在集群上均匀分布边，进而更均匀地平衡整个群集的数据。这个

想法出自 2005 年的研究，这个想法很受图处理系统 GraphLab（现在叫 PowerGraph）的欢迎，也被 GraphX 采用并作为默认的分区方案。

GraphX 支持边上的 4 种不同的分区方案，这将在 9.4 节中进行介绍。GraphX 的顶点分区独立于边的分区。通过避免把一个“高度”的顶点载入单个机器节点，GraphX 避免了早期图处理系统遭受的负载不均衡问题。

图的幂律

已经发现图实际上服从幂律分布。在排名的场景中，通过度（直观来说是受欢迎程度）来看一下那些流行的顶点：最受欢迎的顶点，比第二流行的顶点的流行度高 40%，而第二流行的顶点又比第三热门的顶点的流行度高 40%，依此类推。



在这个排名的场景里，这也被称为齐夫定律（Zipf's Law）。这些都是图的现实情况，图数据的分布是通过顶点切分策略在集群上均衡图数据。Spark GraphX 默认采用顶点切分策略。

1.3.3 GraphX 允许选择：图并行还是数据并行

正如我们看到的，GraphX 在一个表中存储图的边，在另一个表中存储顶点。这使得在 GraphX 中实现的图算法可以高效地遍历图，要么通过图沿着边从一个顶点到另一个顶点，要么通过边或者顶点的表，如图 1.11 所示。后者的访问模式允许高效地批量转换边或顶点数据。

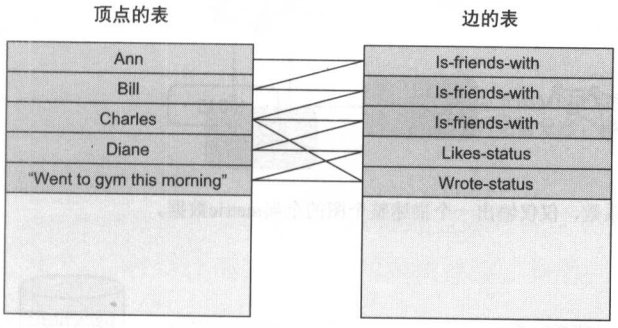
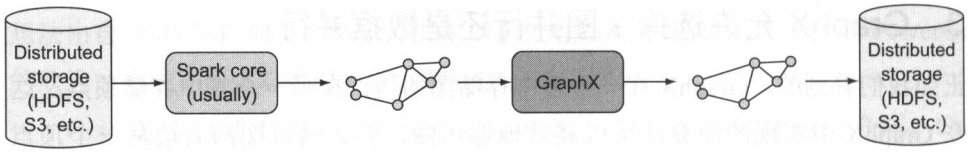


图 1.11 GraphX 加快了通过图并行操作或数据并行操作的数据访问。

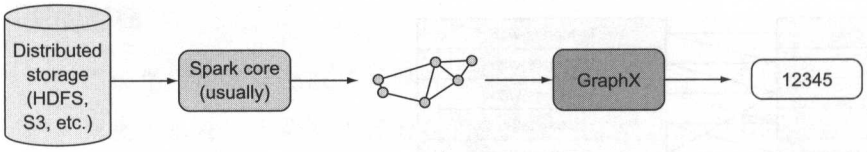
GraphX 将边和顶点保存在不同的表中，虽然也可以用关系型数据库表来做，但 GraphX 内部有专门的索引来快速遍历图，并且提供了 API，使得图查询和处理比用 SQL 更容易。

1.3.4 GraphX 支持的各种数据处理方式

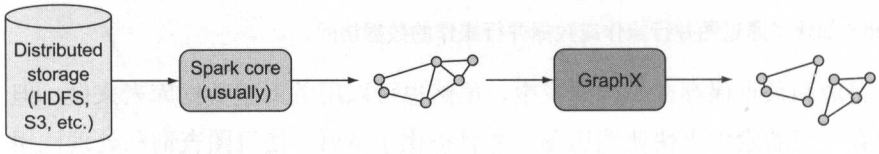
GraphX 内部是一个批处理系统，并没有融入 Spark Streaming（至少不以任何直接方式）。没有千篇一律的方式来使用 GraphX。GraphX 可以提供许多不同的批处理数据流，在图 1.12 和图 1.13 中展示了这些数据流。



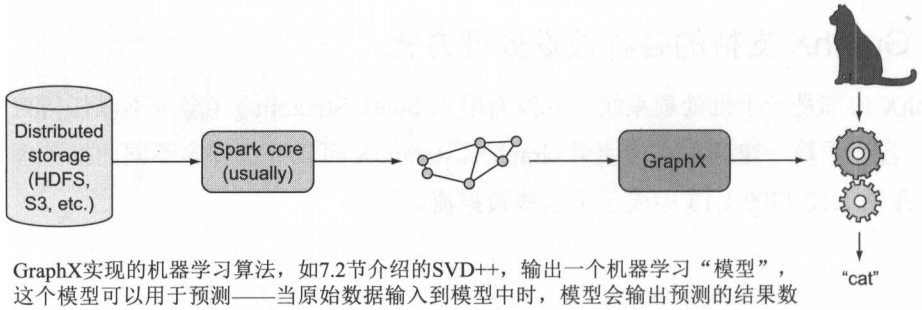
在这个常见的工作流中，图被转换成一个新图（例如，顶点或边可能产生了新的属性值）。类似这种例子可以参阅2.3节和5.1节中介绍的PageRank算法示例。



一些算法，如8.4节介绍的全局聚类系数，仅仅输出一个描述整个图的全局metric数据。



另外一些图算法，如5.4节介绍的连通组件，输出子图。



GraphX实现的机器学习算法，如7.2节介绍的SVD++，输出一个机器学习“模型”，这个模型可以用于预测——当原始数据输入到模型中时，模型会输出预测的结果数据或者标签。

图 1.12 各种可能的 GraphX 数据流。因为 GraphX 读取图数据文件的能力是有限的，通常的数据文件需要用 Spark 的核心 API 转换成 GraphX 需要的图数据格式。GraphX 算法可以输出为另一个图、数字、一些子图或机器学习模型。

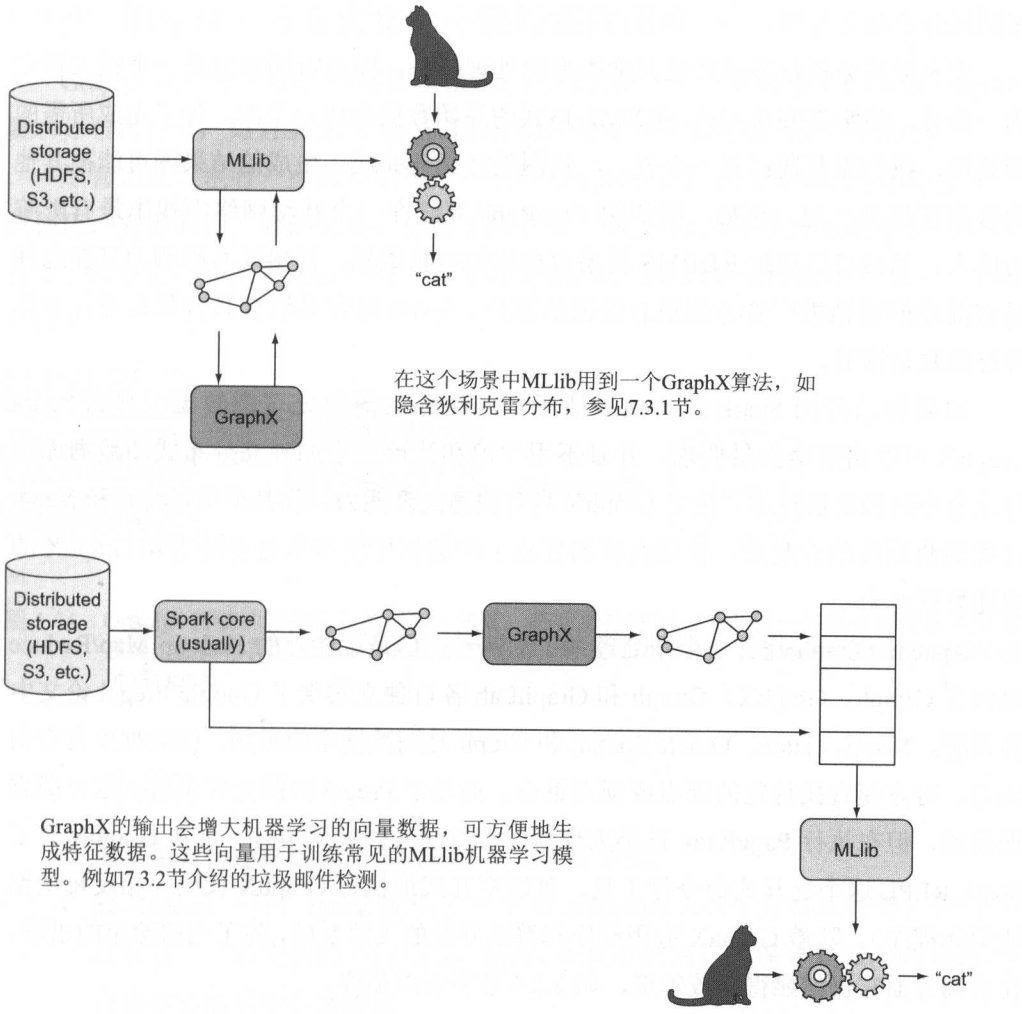


图 1.13 用到 Spark 机器学习组件 MLlib 的数据流，两个场景中的算法都用到 GraphX，GraphX 也可以在任何旧的机器学习算法中使用。

1.3.5 GraphX 与其他图系统

图系统可以分成两个被广泛认可的类别：图处理系统和图数据库。许多图处理系统是基于内存的，有些甚至支持集群计算。Spark GraphX 是一个图处理系统而不是图数据库。图数据库在这些方面有很大的优势：提供数据库事务、查询语言、简单的增量更新和持久化，但是一个基于磁盘的图数据库性能比不上 GraphX 这种全内存的图处理系统。图处理系统很有用，比如响应 Web 服务的请求或者运行一次长

时间运行的独立计算。

大多数图分析任务还需要其他类型的处理任务。图分析通常只是一个较大管道的一部分。经常需要从 CSV 或 XML 格式的原始数据生成一个图。为了生成所需的图属性，我们需要连接另一个表。一旦图处理任务完成，生成的结果图可能被其他的数据连接到一起。例如，我们用 PageRank 算法在一个社交网络中找出最有影响力的人，然后可以通过 RDBMS 关系数据库的销售数据，找出既有影响力又在选择最有前途的营销推广者方面最有价值的客户。Spark 很容易组合具有数据并行和图并行的复杂管道。

如果你已经用 Spark 做其他事情了，同时你也需要处理图数据，那么 Spark GraphX 可以高效地满足要求，并且不用学习和管理一个如独立分布式图数据库这种完全不同的集群技术。由于 GraphX 具有快速处理能力，你甚至可以将它和 Neo4j 这类图数据库结合起来，使用两者的优点：图数据库的事务处理能力和 GraphX 的快速处理能力。

Apache Giraph 是另一个图处理系统的例子，但是性能差的 Hadoop Map/Reduce 限制了 Giraph。GraphX、Giraph 和 GraphLab 各自独立实现了 Google Pregel 论文中的设想。Neo4j、Titan、Oracle Spatial 和 Graph 是图数据库的示例。图数据库有查询语言，可方便查找特定的顶点或顶点集合。而基于 Pregel 的图处理系统在这方面表现很差，但在执行 PageRank 这类大规模并行算法方面表现很好。现在 GraphX 有了 Spark REPL 这个交互式命令行工具，可以交互式地执行 GraphX 代码（相反每次都要编译程序）。随着 GraphX 应用程序和算法开发的飞速发展，除了当前给定的语法，在查询方面它仍然显得比较笨重，如 3.3.4 节所示的那样。

GraphX 出现的时间较晚，它的一些局限性来自于 Spark 本身的限制。例如，GraphX 数据集和其他所有的 Spark 数据集一样不能被多个 Spark 应用程序共享，除非使用 REST Server 内置的一个 Spark JobServer。后来 IndexedRDD 被添加到 Spark 里（Jira ticket SPARK-2365），IndexedRDD 实际上是 RDD 的一个可变（可更新）HashMap 版本，之前 GraphX 受限于 Spark RDD 的不变性，而 RDD 这个数据不可修改的特性对处理大的图是一个问题，现在有了 IndexedRDD，这个问题解决了。GraphX 虽然在一些方面比之前快了，但由于 GraphX 依赖于 JVM，仍然比 GraphLab/PowerGraph 这类用 C++ 写的系统要慢些。

1.3.6 图存储：分布式文件存储与图数据库

GraphX 是一个严格的内存处理系统，所以需要一个存储图数据的地方。Spark 要求分布式存储，如 HDFS、Cassandra 或 S3，分布式存储是通常采用的方式。

尽管如此，也有一些这样的用法，组合 GraphX 这个图数据系统与图数据库一起使用，充分利用二者的长处，如图 1.14 所示。虽然 GraphX 与 Neo4j 的优劣对比有激烈的争论，但是要认识到对一些使用场景，将二者结合使用比单独使用一个要好很多。

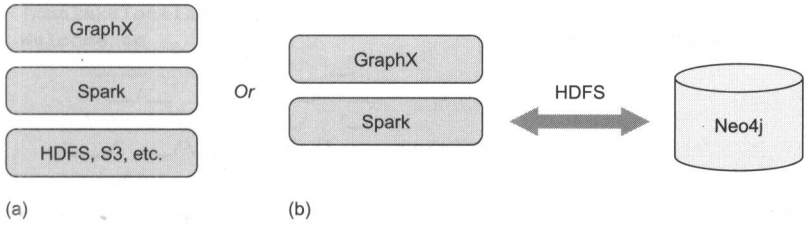


图 1.14 GraphX 传统的和目前最常用的方法是，将数据存储到 HDFS 或一些其他的分布式存储系统。但如果再利用图数据库的成熟能力，可实现两全其美：事务用图数据库，图处理用 GraphX。

1.4 小结

- 图是模型化连接数据的自然和强大的使用方式。
- 和 Hadoop 一样，Spark 提供了一个 Map/Reduce API（分布式计算）和分布式存储。二者主要的不同点是，Spark 在集群的内存中保存数据，而 Hadoop 在集群的磁盘中存储数据。
- GraphX 在 Spark 基础之上提供弹性、高效的图并行处理。
- Spark 还提供了数据并行处理，能比较理想地解决现实世界中通常要求兼顾图并行和数据并行处理的大数据问题。
- GraphX 不是图数据库，不适合查询图的单个顶点或者一部分顶点。GraphX 是图处理系统，适用于 PageRank 这类大规模并行算法。
- 图数据的类型包括网络结构数据、树状结构数据、关系型数据、Kitchen sink 和稀疏矩阵。
- 图算法包括 PageRank、推荐系统、最短路径、社区发现等。

2 GraphX快速入门

本章要点

- 准备练习用的图数据
- 用 Spark Shell 开启学习 GraphX 的第一步
- 使用 PageRank 算法

交互式的 Spark Shell 是快速使用 Spark 最简易的方式，也是挖掘图数据的重要方式。不用编译代码意味着你可以专注于运行程序代码片段并立即查看输出结果。虽然 Spark Shell 使用 Scala 语言编写代码，但如果你之前没用过 Scala 也不用担心，本章会引导你一步步进行开发。

本章的目标是逐步介绍如何使用 GraphX，不深入研究细节。你可以下载一些参考文献部分列出的样本图数据。使用 Spark Shell，运行一些代码，很快你就会确定哪些论文被引用得最频繁。更有趣的是，你将会学习调用 GraphX 内置的网页排名算法在图中找到“最具影响力”的论文。在随后的章节中，我们再深入了解网页排名算法。

2.1 准备开始并准备数据

一般情况下运行一个 Spark 任务的过程是这样的，用 Scala（Java 或者 Python）

写出 Spark 程序，编译，提交任务到 Spark 集群。更便捷的是在交互式命令行工具 Spark Shell 上写一些 Spark 程序，快速验证你的想法。

首先安装好 Spark（可参见附录 A），输入：

```
spark-shell
```

假定 spark/bin 已经在 PATH 中设置了（可以在 ~/.bash_profile 下设置）。如果还没有在 PATH 中设置，那就要打开 Spark 解压后的目录，进入 bin 目录，执行脚本 ./spark-shell，你会看到如下输出：

```
[mmalak@localhost bin]$ ./spark-shell
Welcome to                              <-- 许多日志输出没有显示
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.      <-- 此处省略更多日志输出

scala>
```

在上述日志输出的末尾，Spark Shell 提醒变量 sc 可以直接使用。Spark Shell 启动时在上下文中创建一个 org.apache.spark.SparkContext 对象的实例 sc。SparkContext 引用使用 Spark 的入口，后续会看到它提供了很多 Spark 函数，本章我们会用 SparkContext 将数据加载到 Spark 中。

第二步是准备一些 Spark 要用到的数据。可以用你自己的数据，也可以从斯坦福网络分析项目（SNAP）中下载一些数据，项目的网址为 http://snap.stanford.edu/data。

我们选用 Arxiv-HEP-TH（高能物理理论）引用网络数据集（不要和合作网络数据集混淆），具体下载页面为 http://snap.stanford.edu/data/cit-HepTh.html。下载 1MB 左右的压缩文件 cit-HepTh.txt.gz，解压后为 6MB 左右。文件的开头如下所示：

```
# Directed graph (each unordered pair of nodes is saved once): Cit-HepTh.txt
# Paper citation network of Arxiv High Energy Physics Theory category
# Nodes: 27770 Edges: 352807
# FromNodeId    ToNodeId
1001            9304045
1001            9308122
```

```
1001        9309097
1001        9311042
1001        9401139
1001        9404151
1001        9407087
1001        9408099
```

注释行以#开始，每一行数据表示图中的一条边，每条边有源顶点ID和目标顶点ID。在这个示例中，每个顶点ID都代表压缩文件cit-HepTh-abstracts.tar.gz中列出的指定物理论文。你可以把这个论文压缩文件下载下来，对照一下上述列出的顶点ID和实际对应的论文。在论文引用的这个场景中，源顶点表示较新的论文，目标顶点表示时间较久的论文，新论文引用旧论文。

这是GraphX公认的文件格式。

> **注意**：另外一个主要的标准图文件格式是资源描述框架（RDF），还有N3（Notation 3）这类衍生格式。截至Spark 1.6，GraphX还没有支持RDF文件格式。第8章会演示如何读RDF，现在我们还是先用简单的顶点组成的边列表的格式。

2.2 用Spark Shell做GraphX交互式查询

现在我们来用Spark Shell加载和查询HEP-TH数据集，只需几行代码就能找出引用频繁的论文。加载的数据集可以常驻内存，下面就来演示如何在这个数据集上做进一步分析。交互式查询是Spark的关键特性之一。

> **定义**：Spark Shell是REPL的一个实例，REPL即：读取—求值—输出，循环上述过程。REPL是一个交互式命令行工具，读取输入的每一行代码，立即执行（求值），并在控制台输出执行结果。Scala、Python和其他语言一般都有REPL工具，Spark REPL也是在Scala REPL基础上衍生的。第一个交互式命令行工具在1960年由编程语言LISP实现，Read、Eval、Print和Loop是LISP语言的四个保留函数名。

为了避免路径问题，我们复制cit-HepTh.txt文件到`spark-shell`脚本所在的`bin`目录：

1. 复制 cit-HepTh.txt 文件到 spark-shell 所在目录。
2. 执行 ./spark-shell。
3. 现在在 Spark Shell 里执行下面三行代码，我们就可以找到引用最多的论文：

```
import org.apache.spark.graphx._
val graph = GraphLoader.edgeListFile(sc, "cit-HepTh.txt")
graph.inDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
```

下面来逐行解释：

```
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
scala>
```

Scala 小贴士：Scala 用下画线 _ 作为通配符，这里的下画线 _ 在其他的上下文中可能表达其他的意义。

Scala 的依赖导入和 Java 类似，只不过 Scala 用下画线 _ 表示通配符，Java 中用星号表示通配符。输入一行代码后，REPL 在下一行输出执行结果。这行依赖导入代码，REPL 输出显示确实导入了依赖。如果代码执行报错，也会有反馈输出，看第二行代码的执行：

```
scala> val graph = GraphLoader.edgeListFile(sc, "cit-HepTh.txt")
14/12/14 23:04:06 INFO MemoryStore: ensureFreeSpace...   <-- 省略一些日志输出
graph: org.apache.spark.graphx.Graph[Int,Int] =
 org.apache.spark.graphx.impl.GraphImpl@15721cbd
scala>
```

GraphLoader 是 GraphX 类库里的对象，通过上一步依赖导入。GraphLoader 有一个函数 edgeListFile，用于加载边列表格式的文本文件，生成图。edgeListFile 函数有两个必填参数，第一个是前面 Spark Shell 创建的 SparkContext 参数 sc，第二个参数是边列表文件的路径。我们把 cit-HepTh.txt 文件复制到了 Spark Shell 的当前目录，所以文件路径就是当前目录的文件名。

Spark Shell 的输出表示已经成功创建了一个类型为 org.apache.spark.graphx.Graph[Int,Int] 的对象，变量名为 graph。图 2.1 进一步详细剖析了这行代码。

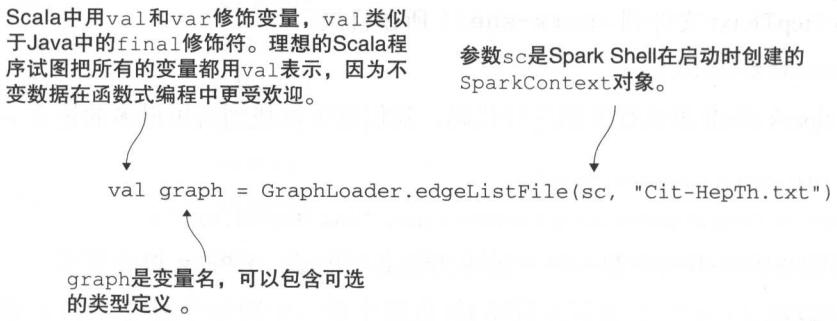


图 2.1 根据边列表格式的数据文件来创建图对象。

> **注意**：graph 是变量名，但是变量对应的类型声明在哪里呢？Scala 是一个采用类型推导的静态类型语言。Scala 编译器能通过函数 edgeListFile() 的返回类型推导出 graph 应该是什么类型，其返回类型是 org.apache.spark.graphx.Graph。一旦编译器确定了 graph 的类型，就不能再修改了。因为这种代码简洁性，Scala 看起来像是种和 Perl 一样的解释型语言，实际上不是。Perl 的变量可以在运行时改变类型，Scala 是严格的静态类型语言，只不过看起来不那么啰唆而已。

Spark 把 HEP-TH 的图数据加载到内存中，现在我们详细地分析一下最后一行代码：

```
graph.inDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
```

图对象调用的 inDegrees 函数可得到一个顶点 ID / 入度对的 VertexRDD[Int]（注意，Scala 中如果函数的参数列表为空，可以省略括号），现在我们可以把这个 RDD 视为一个数组。

RDD 提供了一个 reduce() 函数，该函数只有一个参数，这个参数本身也是一个函数 f:(T, T)=>T，即输入两个参数输出一个结果。引用这个函数就可以把 RDD 中的每两条数据归并成一个结果，归并的规则就是取入度大的顶点，如此归并下去直到只有一条数据留下，最后返回。

不用单独定义一个函数提供给 reduce 函数作为参数使用，实际上我们用的是 Scala 的匿名函数（见图 2.2）。下一章会进一步了解匿名函数的用法，在这里只需要知道函数可以作为参数传递到其他函数中，而且是无须声明的内联函数的形式。reduce 函数执行完后，整个 RDD 包含的（VertexId，in-degree）数据被归并（Reduce）为一条入度最大的数据。

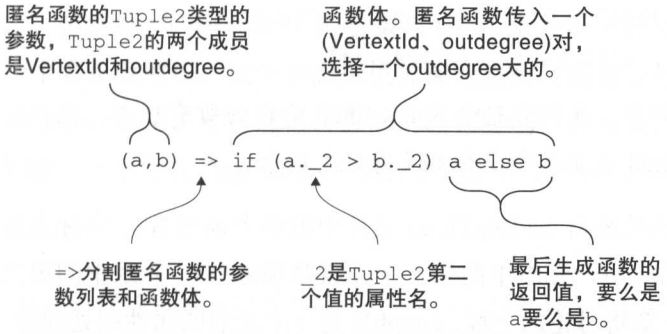


图 2.2 定义一个匿名函数。

当第三行代码输入到 Spark Shell 后，输出被引用最多的一个理论物理论文 ID：

```
scala> graph.inDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
14/12/14 23:50:56 INFO SparkContext: Starting job: ...
14/12/14 23:50:59 INFO SparkContext: Job finished: reduce at
 <console>:18, took 3.16079562 s
res0: (org.apache.spark.graphx.VertexId, Int) = (9711200,2414)

scala>
```

很多日志项没有显示

正如我们所见，确切地说，ID 为 9711200 的论文（1997 年 11 月的第 200 个论文）被其他论文引用了 2414 次之多。

我们可以只用三行代码就计算出引用最多的论文，其中一行还只是导入依赖，另外两行也可以合并成一行，表达力强吧！

这个例子其实没有用到图计算的最强大的功能，它也可以用 SQL 在关系数据库上用一个 GROUP BY 查询出引用最多的论文。在下面的章节中我们将演示在这个相同数据上运行 PageRank 算法，你会看到 GraphX 的强大之处。

2.3 PageRank算法示例

在这一节，你会了解到用图处理方式运行一个最著名的 PageRank 算法是多么容易。虽然 Google 的拉里·佩奇发明的 PageRank 算法（以他的名字命名）只是用来对互联网上的 Web 页面做排名，但这个算法也可以用来度量（计算）图中顶点的影响力。在把 PageRank 算法应用到理论物理引用网络之前，要注意隐藏在使用场景背后的东西。我们先来看一下图中的顶点：

```
scala> graph.vertices.take(10)
res2: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((9405166,1),
```

```
(108150,1), (110163,1), (204100,1), (9407099,1), (9703222,1),
(9709148,1), (9905115,1), (103184,1), (211245,1))
```

注意：在本书其余部分，我们会忽略 Spark Shell 输出的所有日志。我们假定本书中的 Spark Shell 交互式命令行都没有日志输出。

你或许认为图中的顶点还是来自 cit-HepTh.txt 文件中的单个顶点 ID，实际上它们是 Scala 里包含两个元素的元组，元组中的第一个元素是顶点 ID，元组中的第二个元素全部是数字 1。如第 3 章将讨论的一样，GraphX 是一个天然的属性图处理器，这意味着 GraphX 允许顶点和边都有自己的属性集。这些数值 1 是后面用到的函数 `GraphLoader.edgeListFile()` 给顶点属性赋予的一个默认值，所以这些顶点就有了默认值，虽然这个默认值不表示任何意义。GraphX 可以处理有属性值的顶点，而边列表文件中并没有属性值。

从前面的图中可以看到，`Graph` 的 `pagerank()` 方法返回一个新的 `Graph`，其中每个顶点有 `Double` 类型的属性值，这个属性值是 PageRank 算法为顶点计算出的。Spark 中有一个关键机制：已经存在的图不能被更新。所以，在一个图上做一些转换操作后，会直接生成一个新图，如图 2.3 所示。

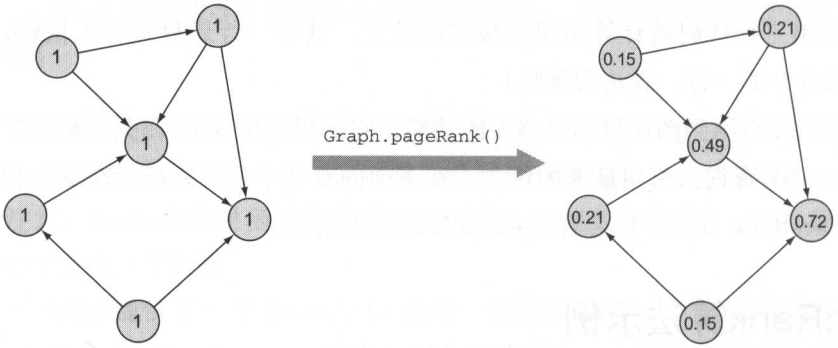


图 2.3 运行 pageRank() 函数，创建出一个新图，其顶点的属性值是 PageRank 算法计算出的权值。

`pageRank()` 函数做了哪些事情？它给每个顶点赋予一个 `Double` 类型的属性值，这个数值能衡量这个顶点在整个网络中的影响力。第 5 章将详细讲述 PageRank 算法的原理，现在我们看看用 GraphX 在一个图上运行 PageRank 算法是多么容易。

```
scala> val v = graph.pageRank(0.001).vertices
v: org.apache.spark.graphx.VertexRDD[Double] = VertexRDD[1264] at RDD at
 VertexRDD.scala:58
```

传递给 pageRank() 的参数值 0.001 是为了平衡速度和最终结果准确度之间的一个容忍度数值。这个容忍度值如果设置得过高则运行速度会很快，但是计算结果不准确，如果设置得过低，算法的运行时间将会太长，精确度没什么变化。现在我们看一下运行完 PageRank 图算法后排前 10 的顶点：

```
scala> v.take(10)
res3: Array[(org.apache.spark.graphx.VertexId, Double)] =
 Array((9405166,1.336783076434938), (108150,0.5836164464324066),
 (110163,0.15), (204100,0.19080382117882116),
 (9407099,0.8271044254712047), (9703222,0.16521611205688394),
 (9709148,0.22176221523472583), (9905115,0.38267418598941183),
 (103184,0.20437621370972553), (211245,0.2298299371239072))
```

这些浮点值就是 PageRank 值，前 10 的 PageRank 范围是从 0.15 到 1.34。

现在我们在结果集 v 上运行 reduce() 归并函数，找出 PageRank 值最高的顶点。

```
scala> v.reduce((a,b) => if (a._2 > b._2) a else b)
res4: (org.apache.spark.graphx.VertexId, Double) =
  (9207016,85.27317386053808)
```

通过 PageRank 算法，我们发现 ID 为 9207016 的论文是最有影响力的论文。如果你已下载了论文摘要的归档文件，找到论文 ID 为 9207016 的论文摘要，这篇论文名为《非对称弦理论》，由 Jnan Maharana 和 John H. Schwarz 在 1992 年发表。

在第 5 章我们将会进一步讲解 PageRank 算法的细节。

2.4 小结

- 用 Spark Shell 交互式查询方式快速挖掘图数据。
- 许多现有的以边列表格式存储的图数据集可以用 GraphLoader.edgeListFile 直接加载。
- 用 graph.pageRank() 函数运行 PageRank 算法。
- 图在 GraphX 中是不可变的，所有图的转换都会返回一个新创建的图。

3 基础知识

本章要点

- Scala 的基本思想，函数式编程和类声明这样的基础知识
- Spark RDD，常用的 RDD 操作，序列化和 SBT 入门
- 图相关的术语

要具有 Spark、Scala 和图的一些基础知识才能较好地使用 GraphX。本章主要介绍这三种技术的基础知识，了解了这些基础内容，才能快速通读本书。

Scala 是一门复杂的不容易上手的编程语言。本书表面上不需要 Scala 知识，但具有一些 Scala 的知识会很有用。Scala 的基本需求都包含在本章的第 1 节中，Scala 的小贴士分布在本书的其余部分，有助于初级和中级的 Scala 程序员进行学习。

本章的第 2 节是 Spark 快速入门教程，想要全面深入学习 Spark，推荐阅读 *Spark In Action*（Manning, 2016）这本书。虽然 Scala 的函数式编程理念被带到了 Spark 中，但 Spark 不像 Scala 那么复杂，所以本书中的 Scala 提示比 Spark 提示多。

最后，对于图，本书不会展开到需要数学公式证明的图论知识。我们前面提到过结构化的属性图，这一章会解释相关的术语定义。

3.1 Scala——Spark的原生编程语言

除了为支持非 Scala 的 API 外，包括 GraphX 在内的 Spark 几乎所有的地方都是用 Scala 实现的。所以，本书除了 10.2 节是以非 Scala 语言举例外，其他所有的地方都以 Scala 来举例说明。本节是 Scala 快速入门，结合贯穿本书的 Scala 小贴士，Scala 和 Spark 的入门知识足够了。Scala 是一门内容丰富且具有极深内涵的语言，完全掌握它或许要花费数年时间，需要慢慢积累。

函数式编程思想对 Spark 的设计及其运作原理有极深的影响。对 Java 或 C++ 程序员来说，Scala 的语法大都比较简单易懂，但有些 Spark 常用的语法结构如类型推导和匿名函数还是需要好好理解的。本部分涵盖了所有的基础内容和 GraphX 用到的概念。

另外，Scala 的复杂性也饱受争议。Scala 提供了强大且简洁的表达能力，但这些优点有时候会被滥用，写出一些不易被他人理解的代码。有些使用 Scala 的项目尝试建立代码规范来规避这个弊端，并且强调要有明确、详细和符合常规习惯的编程风格（追求极致的人会认为这样只能用到少部分语言特性），但发现有些用到的第三方 Scala 类库涉及 Scala 全部特性语法，使用者不得不去适应，这就导致团队中的 Java 程序员生产效率不高。但是对于一些规模小、能力强的团队或者单个程序员来说，Scala 的简洁性能让他们做到高产出。

3.1.1 Scala 的理念：简洁和表现力

Scala 本身就是一门哲学。常会听到有人说 Scala 是一个对象—函数编程语言，这意味着它混合了 LISP、Scheme 和 Haskell 这类函数式编程语言特性和 C++、Java 这类面向对象语言的特性。Scala 融合了这么多特性，凸显出 Scala 设计和使用的两大原则，如下所示：

1. 简洁。做同样的事情，Java 需要 5 行代码，Scala 只需要 1 行代码。
2. 表现力。Scala 的表现力很丰富，表面上看起来像是通过类库添加了关键字和操作符，而不是通过修改 Scala 编译器本身。这就是领域特定语言（DSL），应用的例子包括 Akka 和 ScalaStorm。即使在一些小的方面，Scala 标准库定义的一些函数，看起来就像语言的一部分，如 `&()` 表示求交集（这通常是结合了 Scala 的中缀表达式，看起来像 `A=B&C`，实际上它只是一个函数调用）。

注意：中缀这个术语通常是指运算符位于操作数中间的算术表达方式。例如，在表达式 2+2 中两个数值之间的加号。Scala 有和 Java、Python、C++ 一样常用的方法调用语法：首先是方法名，然后紧随其后的是圆括号括起来的参数列表，如 add(2, 2)。另外，Scala 也有一种特殊的单参数函数的表现形式。

许多 Scala 语言特性（除了函数式编程语言）都很简洁：类型推断、隐式参数、隐式转换的通配符、许多明了的通配符如下画线、case 类、默认参数、局部求值和函数调用时可选的圆括号。我们就不一一列举了，毕竟这不是一本专门讲解 Scala 的书（推荐的 Scala 书，见附录 C）。后面我们会学习类型推断，其他的 Scala 特性如下画线的使用，参见附录 D。一些 Scala 语言的高级特性，在本书中没有全部涉及。下面让我们先来看看什么是函数式编程。

3.1.2 函数式编程

尽管上面提到了多种语言特性，但 Scala 首先且最重要的是一门函数式语言。函数式编程有自己的一套哲学。

- 不变性：函数不应该有副作用（改变系统状态），因为副作用使得它很难在上层对程序执行做推理。
- 函数是一等公民：任何用到如 Int 或 String 的标准类型时，都能用函数实现。并且函数可以被赋值给变量，或者作为参数传递给其他函数。
- 声明式迭代技术：如递归，用于代码中的显式循环。

不可变数据：val

当数据是不可变的，类似于 Java 中的 final 或者 C++ 中的 const，也就没有用于跟踪的中间状态，这就让编译器和编程的概念变得更简单。没有了状态，很多事情也就没有意义了，比如，任何输入 / 输出的顺序都是由它的自然状态决定的。程序员要改掉声明状态可变的变量或集合的习惯，因为在函数编程理念中，可变变量或集合使得程序员更难理解代码逻辑且编译器也很难进行推理。或者更准确地说，函数式编程的程序员知道哪里使用状态和哪里不需要，而相比之下，Java 和 C++ 的程序员可能在有意义的地方恰恰不用关心 final 或 const。在这些方面保持可变状态是合适的：I/O，可实现经典算法或大集合的性能优化。

Scala 用 var 或 val 声明变量，二者仅有一个字母的差别，但意义相差甚远，这也是很多 Scala 或函数式编程的初学者感到困惑的地方，而熟悉 JavaScript 或 C# 的人都知道要使用关键字 var 来声明所有的东西。而 val 声明的变量，必须在声明时就要进行初始化，并且以后不能被重新赋值。var 更像 Java 里的正常的变量声明。按照 Scala 的理念，所有的变量都应该用 val 声明，即使是中间计算过程的变量也应该用 val，只有 val 不能满足的特殊情况才用 var。我们用 Scala 或 Spark 命令行来演示 val 和 var 的用法：

```
scala> val x = 10
x: Int = 10

scala> x = 20
<console>:12: error: reassignment to val
       x = 20
         ^

scala> var y = 10
y: Int = 10

scala> y = 20
y: Int = 20
```

不可变数据：集合

不变量的思想也应用到了集合类里，函数式程序员更喜欢这种不可变的集合。这种不可变的思想在某些场景是很实用的，比如许多小集合，不立即更新的代价很小；还有一些比较理想化的使用场景。有了这种数据不可变的机制，编译器会自动优化掉低效的代码，也可能插入一些可变性来完成数学等价的结果。

Spark 在很大程度上实现了这种不可变的思想，甚至可能比函数式编程系统所实现的还好。Spark 的基础数据集——弹性分布式数据集（RDD）是不可变的。正如本节后面部分提到的 Spark，RDD 上的操作是以惰性方式排队（queued up in a lazy fashion），然后只在需要的时候执行一次，输出最终结果。这样就允许 Spark 系统优化中间操作过程、序列化、磁盘 I/O，以及重新规划网络传输代价大的数据洗牌（shuffle）。

不可变数据：减少副作用

最后我们来探讨一下让函数无副作用这个目标。在函数式编程中，理想化的函

数是给定输入产生输出，任何给定的输入都产生一致的输出，不影响任何全局性的或者传入的参数对象的状态。函数式编译器和解释器可以非常高效地推断出这种无状态的函数，优化这类函数的执行调用。任何事物都无状态是比较理想化的，因为事实上无状态就意味着没有 I/O，但这不现实；除非有充分的理由需要可变状态，无状态化都是一个好的目标。

函数是一等公民

其他语言，如 C++ 和 Java 有指向函数和回调的指针，而 Scala 可以很轻松地声明函数为内联的并可以到处传递它们，不需要像 C++ 和 Java（Java 8 之前）一样声明单独的“类型”或“接口”。

下面来看一下在 Scala 中如何操作，先通过声明函数类型的正常方式来定义一个函数：

```
scala> def welcome(name: String) = "Hello" + name
welcome: (name: String)String
```

函数定义是这样的，首先用关键字 def 声明，接着是函数名，然后是圆括号括起来的参数列表，等号后面是函数体，多行的函数体需要用花括号括起来，单行函数体可以省略花括号。

可以像下面这样调用函数：

```
scala> welcome("World")
res12: String = Hello World
```

正如所料，这个函数返回字符串 Hello World。我们也可以把它定义成匿名函数，传一个其他的值，匿名函数定义如下：

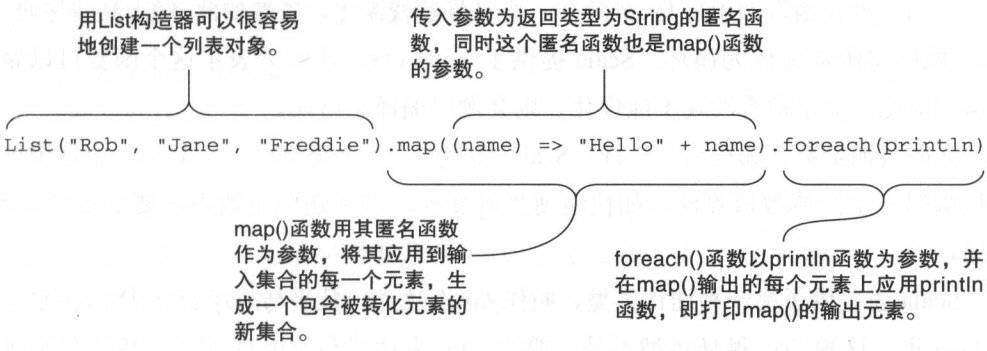
```
(name: String) => "Hello " + name
```

=> 左边定义的是参数列表，=> 右侧是函数体。可以把这个函数赋值给一个变量，然后通过这个变量调用这个函数：

```
scala> var f = (name: String) => "Hello" + name
scala> f("World")
res14: String = Hello World
```

由于可以把函数当成一个有类型的值，上面的匿名函数值的类型是 String =>

String。这样的话，我们可以传递一个函数给另外一个参数有类型要求的函数，如下示例所示，List 调用 map() 函数。



当一个函数引用了一个在本函数外声明的全局或局部变量时，Scala 可以很优雅地处理这种情况。把这些变量包装成一个闭包，如下面的代码所示，Scala 把变量 n 用函数 addn() 包装一下，即使变量 n 在函数 doStuff() 之外值有改变，也是可以的。

```
scala>   var f: Int => Int = null
f: Int => Int = null
scala>   def doStuff() = {
     |     var n = 3
     |     def addn(m: Int) = {
     |       m + n
     |     }
     |
     |     f = addn
     |     n = n + 1
     |   }
doStuff: ()Unit
scala> doStuff()
scala> f(2)
res0: Int = 6
```

迭代式声明，非命令式声明

如果在函数式编程语言里看到了 for 循环，要知道它其实是不推荐使用的，只有在特殊情况下才用。函数式编程语言中两种天然的迭代方式是 map() 和递归。map() 接受一个函数作为参数并将其应用于集合中的每一个元素，这种做法可以追

溯到20世纪50年代的LISP，那时它被称为mapcar（仅晚了FORTRAN语言的DO循环5年）。

递归用来让函数调用自身，风险在于容易导致栈溢出。虽然如此，对于某些递归，Scala能够将函数编译为循环。Scala提供了@tailrec注解来表示这个函数可以做尾递归优化，如果检查发现不能优化，则会抛出编译期错误。

如同其他函数式编程语言一样，Scala也提供了for循环结构，其中一个适用场景是编码一个经典数值算法，如快速傅里叶变换，这种递归函数不需要@tailrec标记。

Scala还提供了另一种迭代类型，叫作*for表达式*，几乎与map()一样。在C++和Java里，这种for循环类型不是必要的。for表达式和map()很大程度上只是风格上不同而已。

3.1.3 类型推断

类型推断是Scala的一大特色，并不是所有的函数式编程语言都有类型推断。在下面的变量声明中：

```
val n = 3
```

Scala根据变量的值3是Int类型的，推断出n对应的类型是Int。这与下面的显式的类型声明是等价的：

```
val n:Int = 3
```

推断出的类型仍然是静态类型，一旦Scala编译器确定了变量的类型，就会一直保持这个类型不变。Scala不是Perl这种动态类型语言，Scala的变量在运行期不能改变类型。类型推断只是为了方便编程。例如：

```
val myList = new ListBuffer[Int]();
```

在Java里，不得不写两次ArrayList<int>类型，一次是变量声明，一次是new对象。注意，Scala里的参数类型声明是用方括号——ListBuffer[Int]，而Java里用的是尖括号。

另外，类型推断也会带来代码混乱不易读的问题。这就是为什么一些团队内部有Scala编码规范来规定类型必须被显式声明。但实际情况是，第三方Scala代码绝大多数依赖类型推断，要确定类型，就要向上追查调用链或由程序员阅读代码了解

内部逻辑。IDE 提供了悬停文本提示来显示推断的类型。

类型推断不容易让人直观看到函数的返回类型，这是因为 Scala 函数的返回类型是由函数最后一行语句的执行结果值决定的（没有 return 语句）。例如：

```
def addOne(x:Int) = {
    val xPlusOne = x+1.0
    xPlusOne
}
```

函数 addOne() 的返回类型是 Double。要是在一个较长的函数里，人眼判别出返回类型需要花些时间。针对这种情况，可以显式声明函数的返回类型：

```
def addOne(x:Int):Double = {
    val xPlusOne = x+1.0
    xPlusOne
}
```

元组

Scala 不支持多个返回值，而 Python 是支持的，但是 Scala 支持能提供相似能力的元组语法。元组是包含了各种类型的值的序列。Scala 中有包含两个元素的二元组类 Tuple2，包含三个元素的三元组类 Tuple3，以此类推，直到二十二元组类 Tuple22。

可以通过成员变量 _1、_2 来访问元组中的元素，数字表示第几个元素，数字从 1 开始。可以这样声明一个元组：

```
scala> val t = Tuple2("Rod", 3)
scala> println(t._1 + "has" + t._2 + "coconuts")
Rod has 3 coconuts
```

Scala 还有一个更简洁的元组声明方式：用圆括号将元组的元素括起来。如下所示：

```
scala> val t = ("Rod", 3)
scala> println(t._1 + "has" + t._2 + "coconuts")
Rod has 3 coconuts
```

3.1.4 类的声明

Scala 里至少有三种声明类的方式。

Java 风格

```
class myClass(initName:String, initId:Integer) {
    val name:String = initName
    private var id:Integer = initId
    def makeMessage = {
        "Hi, I'm a " + name + " with id " + id
    }
}
val x = new myClass("cat", 3)
```

Scala中的变量默认都是public的。

注意，虽然不像 Java 里的显式构造函数，但提供了作为类声明一部分的类参数：在这个例子中是 initName 和 initId。在类代码块中，两个类参数分别赋值给 name 和 id。

在上述代码的最后一行，我们新建了 myClass 的一个实例 x。Scala 里类成员变量默认是 public 的，可以用 x.name 直接访问变量 name。

这样 x.makeMessage 调用 makeMessage 函数，返回一个字符串：

```
Hi, I' m a cat with id 0
```

简写

Scala 的设计目标之一是让代码更简洁，更容易阅读和理解，类定义也遵循这个原则，下面的这个类定义示例就用到了 Scala 两个这方面的特性：

```
class myClass(val name: String, id: Integer = 0) {
  def makeMessage = "Hi, I'm a" + name + "with id" + id
}
val y1 = new myClass("cat",3)      <— 设置 name 为 “cat”, id 为 3
val y2 = new myClass("dog")        <— 设置 name 为 “dog”, id 为 0
```

注意在上例中，用 val 修饰类参数 name，这样 name 就可以被当作类的成员变量使用，而不需要显式地赋值给成员变量。

第二个类参数 id，给它赋了默认值 0，这样只用一个 name 参数就构造了一个对象，也可以两个参数都用。

样本类

```
case class myClass(name:String, id:Integer = 0) {
    def makeMessage = "Hi, I'm a " + name + " with id " + id
}
val z = myClass("cat",3)
```

用样本类新建对象，不需要new关键字声明。

样本类最初是为了这个特定的目的：作为匹配项提供给 Scala 的模式匹配。现在样本类更通用了，与标准的类没有什么不同，除了以下这两点是不同的：构造函数或类中声明的所有成员变量默认是 public 的，以及 equals() 自动被定义好了（也被称为 ==）。

3.1.5 map 和 reduce

看到这个标题，你或许会想到 Hadoop 里 map 和 reduce 的概念（3.2.3 节会详细讲到），但是这个概念也是源于函数式编程（又要追溯到 LISP，不过是用另一个名字表示的）。

举个例子，一个袋子里装满数量不等的水果，要计算出水果总数，用 Scala 可以这样实现：

```
class fruitCount(val name: String, val num: Int)
val groceries: List[fruitCount] = List(
  new fruitCount("banana", 1),
  new fruitCount("apple", 2),
  new fruitCount("orange", 7),
  new fruitCount("pear", 8)
)
groceries.map(f => f.num).reduce((a: Int, b: Int) => a + b)
```

map() 函数将传入的一个转换函数作为参数，把一个集合转变成另外一个集合。reduce() 函数传入一个两两合并的聚集函数作为参数（(Int,Int) => Int），把一个集合聚集为单个值。这个 map-reduce 的过程是：map() 用它的参数函数（fruitCount => Int）把原有类型为 List[fruitCount] 的集合中的所有元素应用一遍，集合变成了 List[Int] 类型，即 List(1,2,7,8)；reduce() 在这个新集合的基础上两两聚集，计算过程如下：

```
1 + 2 => 3
3 + 7 => 10
10 + 8 => 18
```

这种通用的 map-reduce 思想普遍被函数式编程、Hadoop 和 Spark 所采用。

用下画线避免对匿名函数的参数命名

Scala 提供了一个下画线的简写方式代替对 `groceries.map(f => f.num)` 中变量 `f` 的命名，简写如下：

```
groceries.map(_.num)
```

这种写法是有前提条件的，那就是这个参数变量引用的对象只能被用到一次。

_ + _ 用法

_ + _ 是 Scala 的习惯用法，这让很多 Scala 新手感到困惑。虽然这个用法不难理解，但依然是很多人不喜欢 Scala 的有力理由。下画线在 Scala 中一般是通配符，问题是作为通配符的下画线在 Scala 中有十几种不同的用法。第一个下画线表示第一个参数，第二个下画线表示第二个参数，并且这两个参数既没有名字也没有在前面被声明。它是 `(a,b)=>(a+b)` 的简写（其实这个匿名函数本身也是不带类型的简写形式）。这是 Scala 特有的*归并/聚集用法*，一次操作两个数据项。不得不承认，有时需要第二个下画线再次引用第一个参数，一个单参数的匿名函数引用单个参数多次，多参数匿名函数引用多个参数一次，前者比后者更常用。由于这些情况，不得不声明一个变量 `x` 来进行 `x => x.firstName + x.lastName` 操作。Scala 不支持这种匿名函数的简写，所以不得不重新对第二个下画线通配符赋值当前变量来表示第二个参数。_+_ 的适用场景有局限。

3.1.6 一切皆是“函数”

正如我们所见，Scala 函数也会有返回值，返回值是函数体中最后一行代码的（计算）结果值。Scala 里没有“过程（procedures）”，也没有 `void` 类型（虽然 Scala 有与 Java 的 `void` 相似的 `Unit` 类型）。Scala 里万事万物皆是函数，经典的命令控制结构也是用函数实现的。

if/else

Scala 里的 if/else 返回一个结果值，与 Java 中的三元操作符“?”类似，如下所示：

```
val s = if (2.3 > 2.2) "Bigger" else "Smaller"
```

现在把它格式化一下，以便看起来像 Java 代码，但是其内部仍然是函数形式的：

```
def doubleEvenSquare(x: Int) = {
```

```
  if (x % 2 == 0) {
    val square = x * x
    2 * square
  } else {
    x
  }
}
```

在这里，最外层用花括号包含的整个代码块，也就是“then”。if 代码块表面上不是函数式语句，但要记得，这是 doubleEvenSquare() 函数的最后一行语句，所以 if/else 的输出就是函数的返回值。

模式匹配

Scala 中的模式匹配与 Java 中的 switch/case 类似，不同的地方就是函数式。模式匹配会返回一个结果值，采用的是中缀表达式，这与 Java 不同。Scala 的 myState match{case...} 的表示顺序与 Java 的 switch(myState){case...} 正好相反。Scala 的 match/case 比 Java 里的强很多，这是因为 Scala 模式匹配可以匹配 case 样本的数据类型和数据值，这种方式与 Java 不同，Java 是用正则表达式进行模式匹配的（这部分内容不在这里展开）。

如下示例，用 match/case 根据判断浮点数的字符解析器来变换结果状态：

```
class parserState
case class mantissaState() extends parserState
case class fractionalState() extends parserState
case class exponentState() extends parserState
def stateMantissaConsume(c: Char) = c match {
  case '.' => fractionalState
  case 'E' => exponentState
  case _ => mantissaState
}
```

因为 case 类就像结果值，例如 stateMantissaConsume('.') 函数返回了 case 类 fractionalState。

3.1.7 与 Java 的互操作性

Scala 是一门 JVM 语言，所以 Scala 的代码可以调用 Java 代码，同时 Java 代码也可以调用 Scala 代码。此外，Scala 也依赖一些 Java 标准类库，比如：序列化、JDBC、TCP/IP。

Scala 是一门 JVM 语言，同时也意味着 JVM 的通常注意事项对 Scala 也同样适用，即垃圾回收处理和类型擦除。

类型擦除

大多程序员都会涉及垃圾回收问题，这是一个基础问题，而类型擦除比垃圾回收要略微难懂一些。

当在 Java 1.5 中引入范型后，语言的设计者就不得不确定如何实现类型擦除。范型是这样一个语言特性，它允许用户用一个类型来参数化一个类。通常以 Java 集合类举例，对一个集合类添加一个参数：`List<String>`。一旦确定了参数类型，编译器就只允许将 `String` 类型的值添加到集合列表中。

类型信息不会传递到运行期，即类型信息只存在于编译期，也就是说，在运行期，列表的类型仍然是 `List` 而不是 `List<String>`。这种在运行期参数类型丢失的情况被称为类型擦除。如果你写的代码使用或依赖于运行期类型，它可能会导致一些意想不到的和难以理解的错误。

3.2 Spark

Spark 把 Scala 的函数式编程思想引入到了分布式计算领域。这一节，我们将会了解到 Scala 是如何影响了 Spark 的最重要的设计概念：弹性分布式数据集（RDD）。当然本节还会介绍 Spark 的其他特性，熟悉了这些内容后，你就能写出一个合格的 Spark 程序了。

3.2.1 分布式内存数据：RDD

正如第 1 章说到的，RDD 是 Spark 最重要的基础。一个 RDD 就是分布在集群多个节点上的数据集合。RDD 是不可变的，也就是说，已有的 RDD 不能被修改或被更新，但是可以从已有的 RDD 根据转换方法生成一个新的 RDD。一般情况下，

RDD 是无序的，除非已经对它做了 `sortByKey()` 或 `zip()` 这类排序操作。

Spark 有很多种方式从数据源创建 RDD，最常用的方式是 `SparkContext.textFile()`，仅需要一个路径参数：

```
val file = sc.textFile("path/to/file.txt")
println(file.count)
```

textFile()的返回类型是RDD[String]，即包含一行行字符串的RDD。

count函数返回文件中的总行数。

`textFile()` 返回的对象是一个参数化类型的 RDD：`RDD[String]`，文本文件的每一行都被视为 RDD 的一个 `String` 对象。

通过把数据分布在集群中，Spark 可以处理比单机容量更大的数据量，并且可在集群的多个节点上并行处理这些数据。

默认情况下，Spark RDD 存储在集群节点的内存（RAM）中，默认只保留一份，复制因子为 1。这与 HDFS 不同，HDFS 的数据文件存储在复制因子为 3 的集群的多节点的磁盘上（硬盘驱动器或 SSD），如图 3.1 所示。Spark 可以灵活地组合使用内存和磁盘，以及不同的复制因子，这些可以在每个 RDD 运行时进行设置。

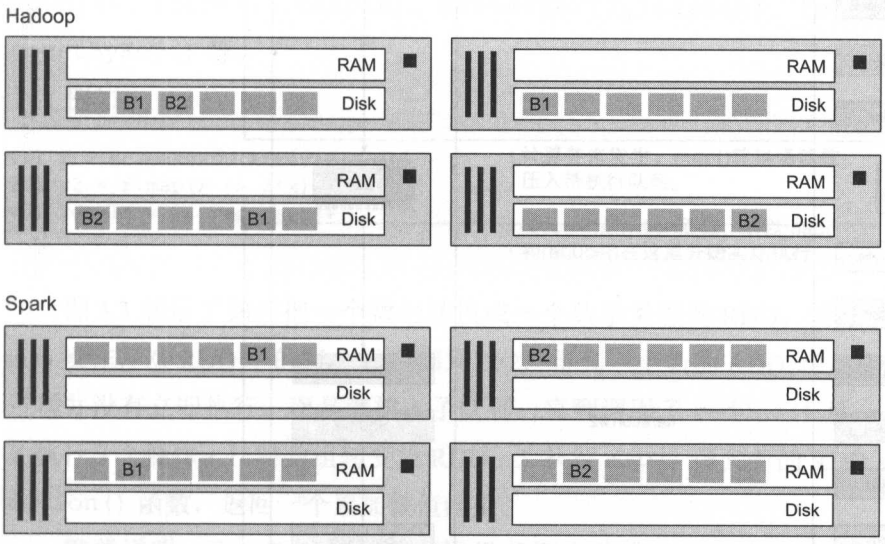


图 3.1 复制因子为 3 的 Hadoop 配置，复制因子为 2 的 Spark 配置。

参数化类型的 RDD 和 Java 集合类类似，并且 RDD 给程序员提供了一个函数式编程风格的 API，如：`map()`、`reduce()` 这两个重要方法。

图 3.2 展示了与 Hadoop MapReduce 相比为何 Spark 比较出色。用在机器学习或者图处理方面的迭代算法，如果用 Hadoop MapReduce 实现，一般需要很重的 Map，

不用 Reduce（称为 map-only 任务）。每一轮迭代结束时都要把这轮的计算结果持久化到 HDFS 中，这就需要一些额外的步骤，如：序列化、解压，这些额外步骤比计算任务本身还要耗时。另一方面，Spark 会把每轮迭代的结果数据都保存在 RDD 中，这就避免了 Hadoop MapReduce 中的序列化、解压等额外消耗，这也是比 Hadoop 快很多倍的原因之一。

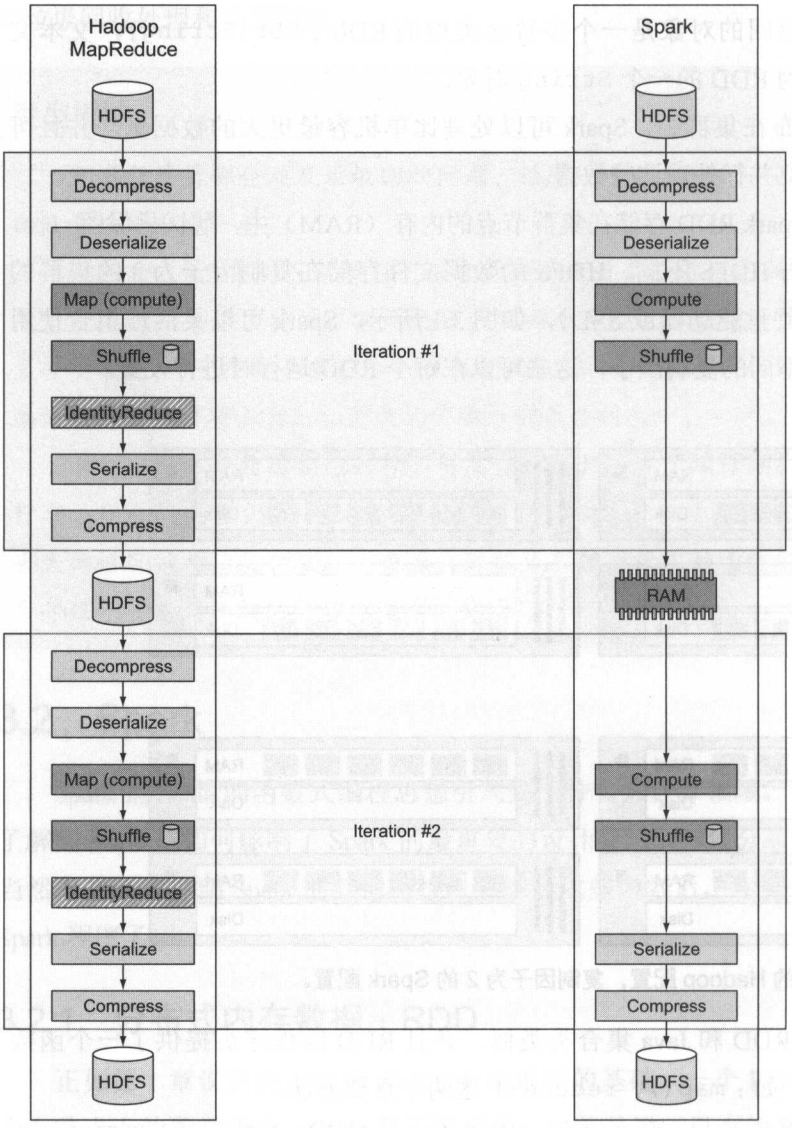


图 3.2 在迭代算法的每次迭代过程中，Spark 会避免 6 次消耗：读磁盘、解压缩、反序列化、序列化、压缩、写磁盘。

3.2.2 延迟求值

RDD 是延迟求值的，RDD 的操作函数只是看起来会被执行，实际上并非如此。RDD 的 API 函数有两类：transformation（转换函数）和 action（执行函数）。Transformation 是延迟执行的，这些函数的调用，会被放入待执行队列，并不会马上执行；当一个 action 函数被调用时，会沿着这个 action 函数向上逐一追溯队列中的 transformation 函数，直到最源头的起始调用（一般是读取数据源），然后再沿着 transformation 函数顺序实际执行，一直执行到 action 函数，得出计算结果。例如，`map()` 是一个 transformation 函数，而 `reduce()` 就是一个 action 函数。Scaladocs 里没有提到这些，transformation 与 action 的文档在 Spark 编程指南中有详细说明：http://spark.apache.org/docs/latest/programming-guide.html#transformations。关键什么是 transformation，一个函数中传入一个 RDD，输出一个新的 RDD，这样的函数就称为 transformation 函数（转换函数）；RDD API 中除了 transformation 函数，其他的可以称为 action 函数。常用的 action 函数如：`reduce(func)`、`collect()`、`count()`、`first()`、`take(n)`、`saveAsTextFile(path)`、`foreach(func)`、`countByKey()` 等。

例如：

```
val r = sc.makeRDD(Array(1,2,3))
val r2 = r.map(x => 2*x)
val result = r2.reduce(_ + _)
```

计算并未发生，map()转换函数被压入待执行队列。

reduce()是action函数，所以map()和reduce()在这里开始实际执行。

图 3.3 展示了如何把一个数组转换成一个数字类型的 RDD。后续对 RDD 进行 map 操作，对 RDD 中的每个元素都乘以 2，得到一个新的 RDD `r2`，但是这个 map 过程并没有立即执行，而是都放入了队列。直到调用了 `reduce()` 后，才会真正开始执行 3 个操作：1. 将数组转变成 RDD；2. 执行 `map()` 得到新的 RDD `r2`；3. 回到 `action()` 函数，返回一个累加数值结果。

需要说明一下，队列这个词在这里并不十分准确，因为 Spark 维护了一个有向无环图（DAG），用于向后追加这些操作。DAG 跟 GraphX 没什么直接关系，由于 GraphX 下面是用 RDD，DAG 和 RDD 是封装在 Spark Core 模块中的，DAG 对 RDD 的处理被隐藏起来以便于对外提供基础的功能。通过维护 DAG，Spark 会避免重复计算多次的问题。

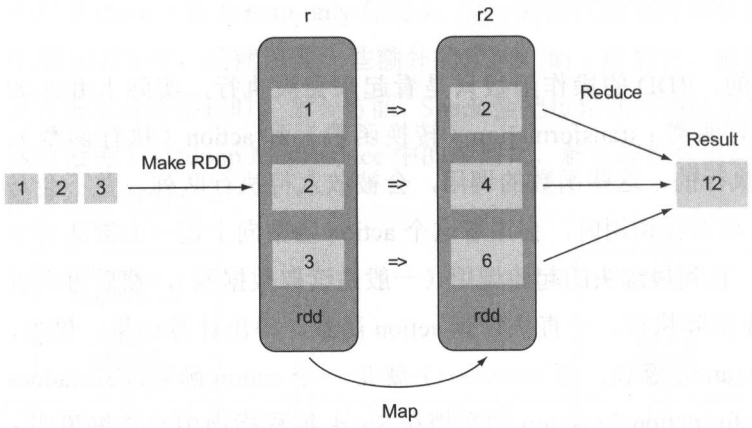


图 3.3 从一个数组构造一个简单的 RDD，对 RDD 进行 map-reduce 操作。

缓存

如果我们用 RDD `r2` 执行另外一个 action 函数，如执行一个 `count()` 函数来计算出 RDD 的数据总条数，会发生什么？如果只这样调用 RDD `r2` 的话，整个 RDD lineage 会从 `makeRDD()` 开始重新被计算一遍。

在 Spark 正在处理的 pipeline 中，在 lineage 里会有很多 RDD，起始的 RDD 一般是从数据源读入数据。一遍又一遍地重新运行相同的数据处理，显然是没有意义的。

为什么不把计算的结果缓存起来呢，免得重新计算？Spark 用 `cache()` 或 `persist()` 来避免这个问题。当调用 `cache()` 或 `persist()` 时，这就表示让 Spark 保留 RDD 的数据不要释放掉，以避免从第一个初始 RDD 到当前这个 RDD 之间的计算过程被重新执行。注意执行缓存实际上是发生在 action 函数被调用后，而不是缓存第一次被使用时。我们把上一个例子修改一下：

```
val r = sc.makeRDD(Array(1,2,3))
val r2 = r.map(x => 2*x).cache
val result = r2.reduce(_ + _)
val count  = r2.count
```

调用了cache()，但并没有产生实际的运算。

调用了action函数reduce()，开始计算。

对r2的元素计数并没有重新计算整个pipeline。

第 9 章会有更多关于什么时候使用 `cache/persist` 以及如何使用的内容。

3.2.3 集群要求和术语解释

Spark 并不是孤立存在的，它也需要其他的应用软件和它一起提供服务，如图 3.4 所示。

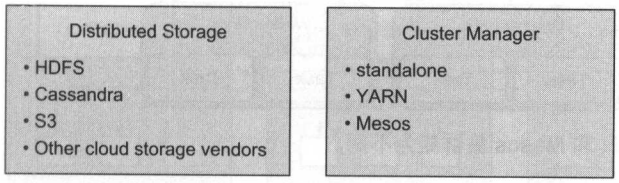


图 3.4 Spark 需要两个主要系统：分布式文件系统和集群管理系统。这两个都是可选的。

正如附录 A 和其他地方提到的，分布式存储和集群管理对于测试和开发环境来说不是必需的。本书中的大多数示例都使用 standalone 模式以及本地存储方式。

各种技术的优点和缺点都不在本书讨论范围内。术语 *standalone* 的意思是 Spark 集群管理使用本地模式。standalone 本地模式一般只给 Spark 使用，Hadoop/YARN 等应用并不能共享使用这些资源。与此相反，YARN 和 Mesos 可以比较方便地在多用户和多应用之间共享重要的集群资源。YARN 更多以 Hadoop 为中心，支持 HDFS 数据本地化（见 SPARK-4352），而 Mesos 更通用并且可以细粒度地管理资源。

图 3.5 所示的 standalone 集群提到的名词术语，可分为以下 4 个方面：

- driver
- master
- worker
- task

driver 是用户所写的业务代码，其中创建的 SparkContext 对象是使用 Spark 的入口，然后提交这些用户代码到有 Master 控制的集群中执行。Spark 调用集群中的机器节点，即 worker，worker 会给每个 task 任务分配一个 CPU（例如，对于一个 8 核机器，worker 就可以分配 8 个 task 任务）。task 任务默认是单线程执行。

最后一个术语是 *stage*。当 Spark 计划好如何通过集群来分布式执行 task 任务时，它会准备好一系列 stage，每个 stage 由多个 task 任务组成，Spark 调度器会决定如何映射这些 task 任务到 worker 节点上执行。

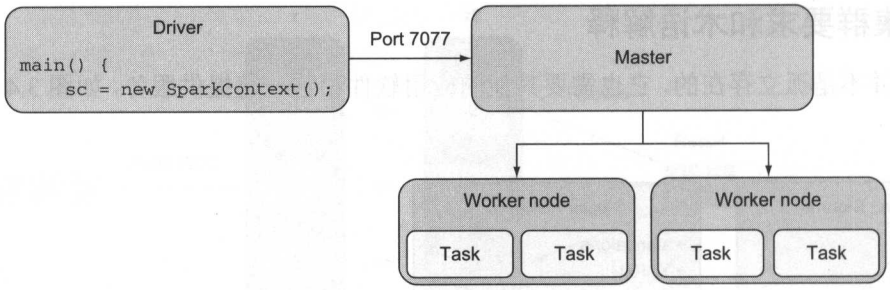


图 3.5 standalone 集群的术语，YARN 和 Mesos 集群略为不同。

3.2.4 序列化

Spark 在 driver、master、worker 和 task 之间传输数据之前，首先要序列化数据。例如，如果要用 `RDD[MyClass]`，首先要保证 `MyClass` 可序列化。最简单也是最早使用的方法是继承 `Serializable` 接口（`Serializable` 是 Java 接口，实现一个接口，在 Scala 里用 `extends` 继承代替 `implements`）。

有两个比 `Serializable` 的性能更好的可选对象序列化方式，一个是 Kryo，另一个是 `Externalizable`。Kryo 比 `Serializable` 更快，压缩也更高效。Spark 内置了对 Kryo 的支持，但也有不少问题，在 Spark 1.1 等早期版本中，有许多 Kryo 支持方面的 Bug，就算是在 Spark 1.6 中，仍然有很多关于边样本类的 Jira 问题未解决。

`Externalizable` 允许你定义自己的序列化方法，例如直接调用 Avro 这类序列化 / 压缩类库，这是一个比 `Serializable` 的性能和压缩更好的合理选择，但这需要大量的样板代码。

本书为了简便使用 `Serializable`，推荐任何在原型阶段的工程都采用这种简单的序列化方式，在需要性能调优的时候可以选择上述两种可选序列化方式。

3.2.5 常用的 RDD 操作

map/reduce

已经看到两个 Spark 的 map/reduced 示例了，下面要进行的会与纯内存的计算不同。数据确实是保留在内存中（默认的存储级别设置），一般情况下许多 transformation 和 action，如 `map()` 和 `reduce()`，通常会有 shuffle 的过程。

shuffle 包括两个部分，一个是写文件到磁盘的 map 任务，另一个是 reduce 任务（参

见图 3.6），reduce 任务要读取数据，如果 map 和 reduce 发生在不同的机器上，就会发生网络传输。

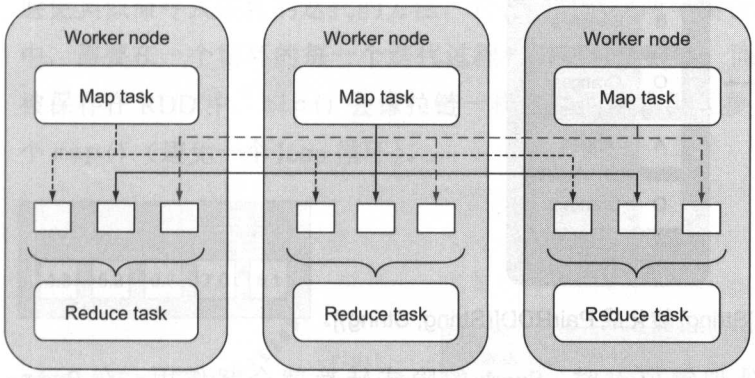


图 3.6 Spark 在 map 和 reduce 之间会有 shuffle 过程，自 Spark 1.6 起，shuffle 总会读写磁盘。

第 9 章会介绍避免和优化 shuffle 的方法。

键值对

标准的 RDD 是数据集合，如 `RDD[String]` 或 `RDD[MyClass]`。另外一个主要的 RDD 是 PairRDD，它提供了对键值对数据的支持。如果从 Scala 的二元组构建 RDD，就会直接自动创建一个 PairRDD。例如，一个由 `String` 和 `Int` 构成的元组集合，对应的 RDD 类型应该是 `RDD[(String, Int)]`（正如前面提到的，在 Scala 中，括号中有两个值就对应于类 `Tuple2`）。

下面是构建一个 PairRDD 的常用方式：

```
val r = sc.makeRDD(Array("Apples", "Bananas", "Oranges"))
val pairrdd = r.map(x => (x.substring(0,1), x)).cache
```

map函数的参数为匿名函数，该匿名函数输出一个(String, String)元组，map函数返回的类型为PairRDD[(String, String)]。

图 3.7 展示了转换的过程。

在 `Tuple2` 中，可以认为第一个值是 key，第二个值是 value，大家能看到这个由 `Tuple2` 组成的 `RDD[(K,V)]` 贯穿 `PairRDDFunctions`，`PairRDDFunctions` 主要是基于 `RDD[(K,V)]` 的一些操作。

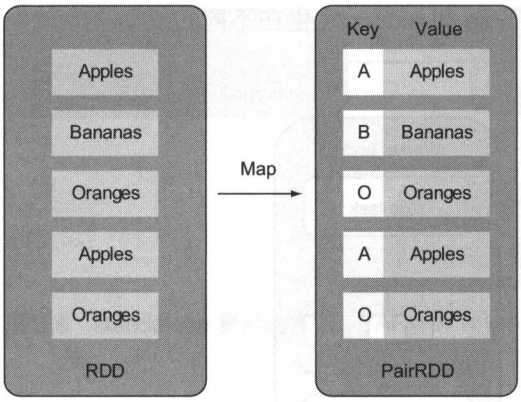


图 3.7 一个简单的 RDD[String] 转化成 PairRDD[(String, String)]。

当你在 RDD 中处理键值对时，Spark 的隐式转换就会起作用。在 PairRDDFunctions 的文档中列出了这些可选的操作函数，如果调用了这些操作函数，Spark 在必要时会自动把一个 RDD 转成一个 PairRDDFunctions 对象。所以，我们可以把 PairRDDFunctions 的函数视为 RDD 的一部分。要开启这个自动转化的功能，必须导入以下类库：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
```

key 可以是 Scala 中的内置类型或者自定义类型，如果用了自定义类型，要确保重载了函数 equals() 和 hashCode()。

PairRDDFunctions 中的许多操作函数是基于通用函数 combineByKey() 的思想的，即将相同 key 的数据项分为一组，然后应用传入的参数（操作函数）。例如，groupByKey() 与 SQL 中的 group by 用法类似；而 Spark 中的 reduceByKey() 的性能更好。第 9 章会讨论性能问题。

PairRDDFunctions 另外一个与 SQL 类似的操作是 join()，两个键值对 RDDjoin() 以后，生成一个新的包含键值对的 RDD，新 RDD 的键值对的 value 值包含了两个 RDD 相同 key 对应的值。

最后一个函数是 sortByKey(),sortByKey() 是对 RDD 中的键值对进行排序，排序并不保证每个操作的结果是一致的。对于 Int 和 String 这类原型类型，排序的结果是可预期的，但是如果 key 是自定义类，你就要定义一个用到 Scala 隐式转换的 compare() 比较函数。

其他有用的函数

zip() 是一个函数式编程中极为有用的函数，也是另一种（还有 map() 和递归）避免类似命令式编程的迭代的方法，它会同时迭代两个集合。要在命令式编程语言中，需要在一个循环的每一个迭代过程中访问两个数组。而在 Spark 中，这些数据被保存在 RDD 中，zip() 会像拉链一样合并两个数组（参见图 3.8），然后执行一个 map()（避免一个 loop 循环）。

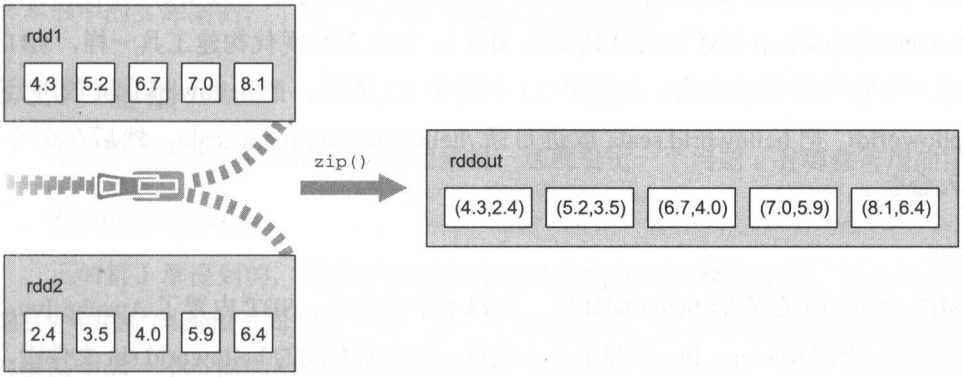


图 3.8 zip() 合并两个 RDD 的数据，随后应用一个 map() 操作。

函数式编程编程中 zip 的一个常用用法是，带整数序列 ID 的拉链合并。对于这种情况，Spark 提供了 zipWithIndex() 函数。

还有两个有用的函数，一个是 union()，它追加一个 RDD 到另外一个 RDD 中（这里的追加未必是追加在后面，因为 RDD 一般不保证数据的顺序，所以未必追加到后面）；distinct() 是另一个有用的函数，它和 SQL 里的 distinct 类似。

MLlib

MLlib 是 Spark 中的机器学习类库组件，除了机器学习部分，它还包括了一些 RDD 基本操作函数，即使你不会用到机器学习也不应忽视这部分 RDD 基础内容。

- 滑动窗口：当需要在一组 RDD 数据上操作，如计算移动平均数（如果你熟悉股票图表中的技术分析）。再比如做一个有限脉冲响应（FIR）的滤波器（如果你熟悉数字信号处理 DSP），mllib.rdd.RDDFunctions 里的 sliding() 函数可以满足上述场景。sliding() 生成 RDD[Array[T]]，其中的数组长度即是指定的窗口长度。在窗口滑动过程中，需要通过指定的窗口长度因子复制内存中的数据为一个 RDD，而这也是最简单的确定滑动窗口规则的方式。

- 统计：RDD通常是一维的，RDD[Vector]则是一个二维矩阵。mllib.stat.Statistics的colStats()函数可以计算二维矩阵RDD[Vector]的列统计信息。

3.2.6 Spark和SBT初步

SBT（Simple Build Tool）全称简单构建工具，是Scala的构建工具，类似Maven。你要是还没有安装SBT，可以从www.scala-sbt.org上下载安装（附录A中的Cloudera QuickStart VM也会默认安装SBT）。如大多数现代构建工具一样，SBT也要求一个特殊的目录结构。如清单3.1和清单3.2所示，把helloworld.sbt放进目录/helloworld，把helloworld.scala放进目录/helloworld/src/main/scala，然后在命令行输入命令：

```
sbt run
```

SBT会自动下载安装Scala编译器，可以不单独安装。SBT内置了Apache Ivy，用于包管理，类似Maven。Ivy来源于Ant项目。在下面看到的helloworld.sbt文件中，libraryDependencies这一行会让SBT去请求Ivy来下载Spark 1.6的jar包及其依赖，并缓存到本地目录~/.ivy2/cache中。

清单3.1 helloworld.sbt

```
scalaVersion := "2.10.5"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0"
```

清单3.2 hellworld.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object helloworld {
  def main(args: Array[String]) {
    val sc = new SparkContext(new SparkConf().setMaster("local")
                                             .setAppName("helloworld"))
    val r = sc.makeRDD(Array("Hello", "World"))
    r.foreach(println(_))
    sc.stop
  }
}
```

创建用到 GraphX 的应用时，就需要引入下面的 GraphX 依赖到 SBT 文件中：

```
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "1.6.0"
```

3.3 图术语解释

本书尽量避免使用图的“理论”，没有涉及边数和顶点数的理论证明。但要了解本书中的实际案例，了解一些术语和定义是有帮助的。

3.3.1 基础

本书中用图来对现实世界的问题建模，这就引出一个问题：有哪些可用的图？

有向图和无向图

正如第 1 章提到的，用图可以对事物以及事物之间的联系建模。首先来弄清楚图 3.9 中的有向图和无向图的不同。在一个有向图中，联系是从源顶点到目标顶点。万维网中的一个页面到另一个页面的链接，学术论文中的引用，都是比较典型的例子。在有向图中，一条边的两个顶点一般扮演着不同的角色，如父子关系，页面 A 链接向页面 B。

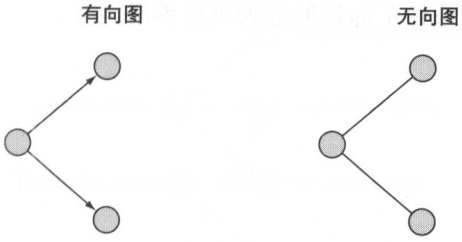


图 3.9 GraphX 中的图都继承了有向图，同时在一些内置算法中也支持忽略方向的无向图。如果需要无向图，可以通过忽略方向来实现。

在一个无向图中，边没有方向；关系都是对称的。社交网络中一个典型的联系是，通常 A 如果是 B 的朋友，我们很可能认为 B 也是 A 的朋友。更进一步，如果我们到 Kevin Bacon 是六度关系，那么 Kevin Bacon 到我们也是六度关系。

GraphX 中的一个重要概念是，所有的边都有一个方向，那么图就是有向图；如果忽略边的方向，就是无向图。

有环图和无环图

有环图是包含循环的，一系列顶点连接成一个环（参见图3.10）。一个无环图没有环。需要了解有环图和无环图的区别的原因是，如果你有一个算法，通过连接的顶点，沿着连接的边，有向图就会造成这样的风险：不恰当的算法实现会卡住，在环上永远循环下去。

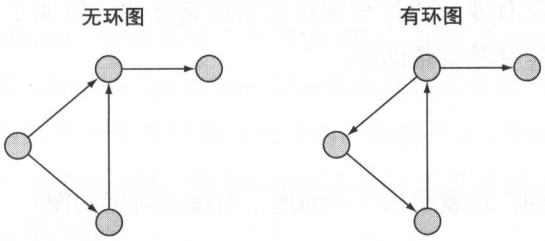


图3.10 一个有环图是在图中存在一个环。在有环图中，如果不关心终止条件，算法可能会永远在环上执行，无法退出。

有环图的一个有趣的特征是，其形成一个三角形关系，即每个顶点都与其他两个顶点相连。三角形关系的用途之一是作为一个预测特征来区分垃圾邮件和非垃圾邮件。

有标签的图和无标签的图

有标签的图是顶点或边除了有唯一标示，还有与之关联的数据（标签）（参见图3.11）。对顶点做了标签的称为顶点标签图，对边做了标签的称为边标签图。

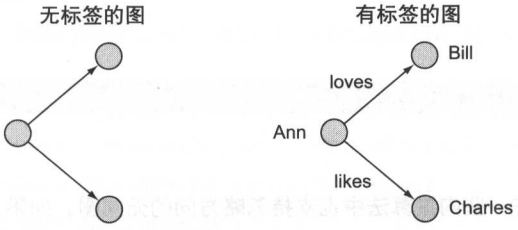


图3.11 一个完全无标签的图通常用处不大，一般至少是顶点有标签。GraphX的基本工具类GraphLoader.edgeListFile支持有标签的图和无标签的图。

在第2章讲到了GraphX用`GraphLoader.edgeListFile()`创建边，它同时会对边的两个顶点（源顶点和目标顶点）创建一个默认属性值1。

一个确定了边标签类型的图也就是大家熟知的权重图。权重图一般会用作计算比如城镇之间的最短路径。这里提到的权重是对边打标签，表示两个顶点（城镇）之间的距离。

平行边和环

平行边和环的区别在于，是否允许在同一对顶点上同时存在多条边，还是边的起点和终点都是同一个顶点，如图 3.12 所示的两种可能性。GraphX 是伪图，会存在平行边和自环，所以需要通过 `groupEdges()` 和 `subgraph()` 来排除这两种情况。

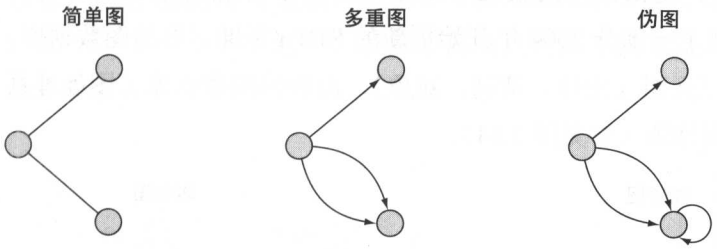


图 3.12 简单图是无向图，没有平行边和环；多重图有平行边；伪图有环和平行边。

二分图

二分图有一个如图 3.13 所示的特定的结构，整个图的顶点被分成两个不同的集合，所有的源顶点是一个集合（所有源顶点之间没有联系），所有的目标顶点是一个集合（目标顶点之间没有联系），在两个集合内都不存在相连的边。

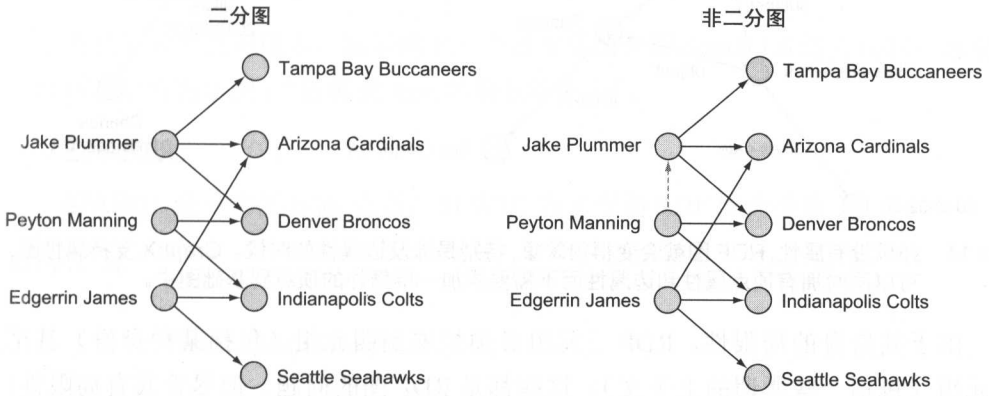


图 3.13 二分图常出现在社交网络分析中，要么如图所示的分组关系，要么是独立的分组关系，如男性和女性群体在异性交往网站的数据分析。一个二分图中所有的边从一个集合指向另一个集合。非二分图不会这样划分，只要有一条边仅仅出现在其中一个集合中，就不是二分图。

可以用二分图对两个不同类型实体之间的关系建模。例如，对于申请上大学的学生，把每个学生划分到一个顶点集合，把要申请的大学划分为另一个顶点集合。

另一个例子是推荐系统，把用户划分在一个集合中，用户所购买的产品划分在另一个集合中。

3.3.2 RDF 图和属性图

资源描述框架（RDF）是由万维网联盟（W3C）在 1997 年首次提出的针对语义 *Web* 的图标准。它实现了一部分 2004 年开始更新的 RDFa 标准。旧的图数据库 / 图处理系统只支持 RDF 三元组（主体、谓词、对象），而新的图数据库 / 图处理系统（包括 GraphX）支持属性图（参见图 3.14）。

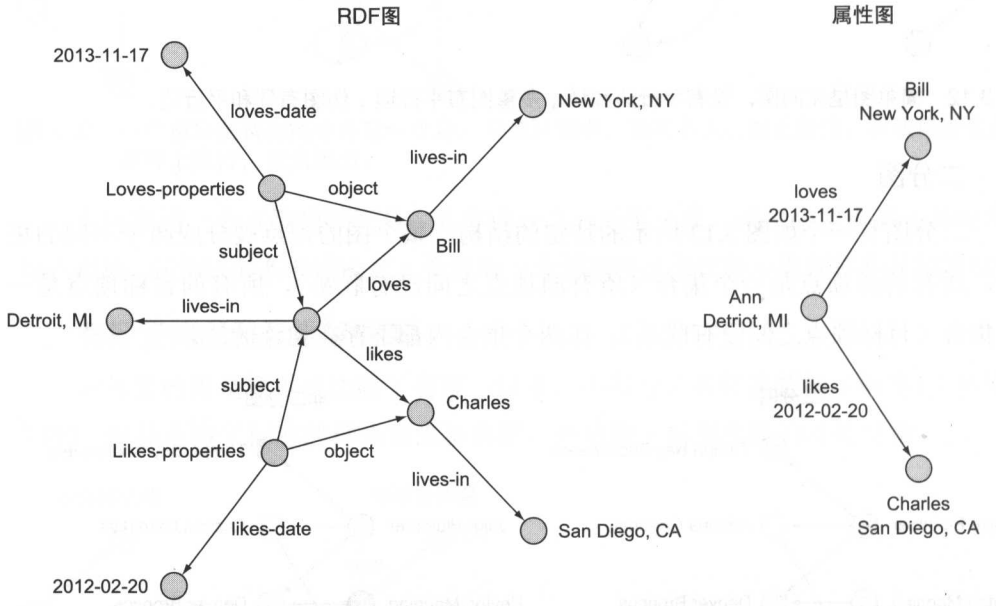


图 3.14 如果没有属性，RDF 图就会变得很笨重，特别是涉及边属性的时候。GraphX 支持属性图，可以同时拥有顶点属性和边属性而不需要添加一堆额外的顶点到基础图中。

由于其自身的局限性，RDF 三元组必须扩展到四元组（包括某种身份）甚至五元组（包括一些所谓的上下文）。这些都是 RDF 图的问题。但尽管其有局限性，RDF 图由于图数据的原因还是很重要的，例如来自维基百科、WordNet 和地理名称的 YAGO2 数据库。

属性图很容易满足新的图数据的需要。

3.3.3 邻接矩阵

另一种表示图论的方式是邻接矩阵(参见图3.15),这不是GraphX表示图的方式。抛开GraphX,Spark的MLlib机器学习库已经支持邻接矩阵,还有更通用的稀疏矩阵。如果你不需要边属性,可以在MLlib找到比GraphX性能更好的算法。例如,推荐系统,从性能的角度来看,`mllib.recommendation.ALS`会是比`graphx.lib.SVDPlusPlus`更好的选择,虽然不同场景适用不同算法。SVDPlusPlus会在7.1节具体介绍。

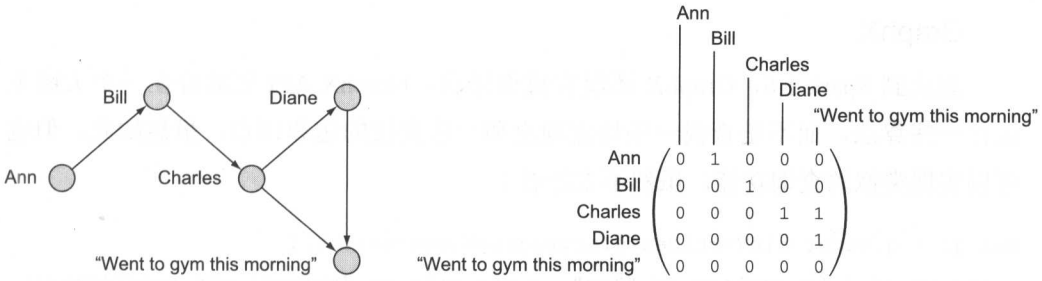


图3.15 一个图和它等价的邻接矩阵。注意,邻接矩阵无法保存边属性。

3.3.4 图查询系统

有几十种图查询语言,这里挑出三个最常用的来跟Spark 1.6进行比较。这里统一使用这个查询示例:“告诉我Ann的朋友的朋友”。

SPARQL

SPARQL是一个类SQL语言,由W3C为了查询RDF图而推出:

```
SELECT ?p
       {
          "Ann" foaf:knows{2} ?p
}
```

Cypher

Cypher是属性图数据库Neo4j使用的查询语言:

```
MATCH (ann { name: 'Ann' })-[:knows*2..2]-(p)
RETURN p
```

Tinkerpop Gremlin

Tinkerpop 努力创建一个图数据库和图处理系统的接口标准，就像 JDBC 一样，只不过比 JDBC 更复杂。Tinkerpop 由多个组件构成，而 Gremlin 是查询系统。有一个把 Gremlin 整合到 GraphX 的尝试，即 Spark-Gremlin 工程，可在 https://github.com/kellrott/spark-gremlin 上查看，截止到 2015 年 1 月，这个工程的最新状态是：没什么要做的了。

```
g.V("name","ann").out('knows').aggregate(x).out('knows').except(x)
```

GraphX

截止到 Spark 1.6，GraphX 还没有查询语言。GraphX API 更适合在一个大图上运行一些算法，而不是查找一些特定顶点和一些直接的边和顶点。虽然如此，但也可以实现类似的查询功能，虽然不太好看：

```
val g2 = g.outerJoinVertices(g.aggregateMessages[Int](
  ctx => if (ctx.srcAttr == "Ann" && ctx.attr == "knows") ctx.sendToDst(1),
  math.max(_,_)))((vid, vname, d) => (vname, d.getOrElse(0)))
g2.outerJoinVertices(g2.aggregateMessages[Int](
  ctx => if (ctx.srcAttr._2 == 1 && ctx.attr == "knows") ctx.sendToDst(2),
  math.max(_,_)))((vid, vname, d) => (vname, d.getOrElse(0))). vertices.
  map(_._2).filter(_._2 == 2).map(_._1._1).collect
```

上面的示例实在有些复杂，在这里就不详解了。在本书的第 2 部分末尾会详细解析这部分代码。这个示例要表达：截至 Spark 1.6.0，GraphX 还没有一个快速方便的查询语言，要实现查询功能比较困难。有两个麻烦的地方：一个是在图中查询指定的节点，另一个是遍历图需要恰好两个步骤（而不是一个步骤，或者由其他条件约束的不限数量的步骤）。

不过也有一些其他工具可以缓解不能查询的问题。在第 10 章，我们会介绍 GraphFrames，这是 GitHub 上的一个类库，提供了 Neo4j 的 Cypher 查询语言的一个子集，与 Spark SQL 一起允许快速方便地对图进行查询。

3.4 小结

- 使用 GraphX 的先行知识：Scala、Spark 和图。

- Scala 是一个面向对象和函数的编程语言，它不仅具有函数式编程的思想，也有自身的特点：简洁性，许多特性在类库中实现而不是语言本身提供。
- Spark 实际上是一个引入了弹性分布式数据集（RDD）的分布式 Scala。
- GraphX 是 Spark 提供的图处理模块。
- 图具有特定的术语。
- GraphX 支持属性图。
- GraphX 没有查询语言，而图数据库有。

第2部分 连接顶点

GraphX 有许多内置算法的 API，但并不是所有的 API 都有详细的文档。第 4 章将引导你如何在 GraphX 中执行基本操作，包括如 `pregel()` 这类基础操作，`pregel()` 是在 Google 的 Pregel 图系统之后出现的。

第 5 章介绍了 GrpaphX 提供的核心算法，如 PageRank，探讨如何用它解决现实问题。

前 5 章涵盖了所有基础工作。第 6 章和第 7 章我们真正开始将 GraphX 付诸实际行动。第 6 章展示了如何在 GraphX 中实现一些经典图算法，如最小生成树。你还将看到一个可用于最小生成树的示例。

机器学习已经变得很普遍。虽然 Spark 的 MLlib 是在 Spark 上进行机器学习的主要方式，但是可以用 GraphX 完成基于图的机器学习。在第 7 章中，你将学习推荐系统（如 Netflix 或 Amazon 所使用的）和文档分类的机器学习的基础知识。然后，你将看到如何在垃圾邮件检测示例中将 MLlib 与 GraphX 组合起来使用。

4 GraphX 基础

本章要点

- GraphX 的基础类
- 基于 Map/Reduce 和 Pregel 的 GraphX 基础操作
- 磁盘序列化
- 常用的图生成工具

迄今为止，我们已经介绍了 Spark 和图的基本知识，现在将二者融入到 GraphX 中。在本章中，你会用到基础的 GraphX API，以及可替换的、性能更佳的 Pregel API。你还会进行图的读写操作，这次没有现成的图数据可用，需要生成随机图。

4.1 顶点对象与边对象

正如第 3 章讨论过的，弹性分布式数据集（RDD）是构建 Spark 程序的基础模块，它提供了灵活、高效、并行化数据处理和容错等特性。在 GraphX 中，图的基础类为 `Graph`，它包含两个 RDD：一个为边 RDD，另一个为顶点 RDD，如图 4.1 所示。

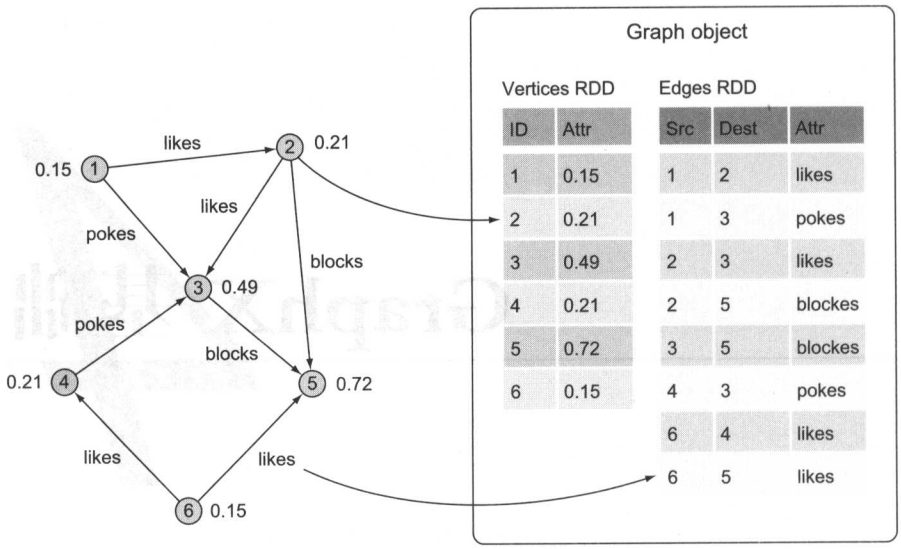


图4.1 GraphX的Graph对象包含两个RDD：顶点RDD和边RDD。

与其他图处理系统和图数据库相比，基于图概念和图处理原语的GraphX，它的一大优势在于，既可以将底层数据看作一个完整的图，使用图概念和图处理原语；也可以将它们看作独立的边RDD和顶点RDD，使用数据并行处理原语，进行mapped、joined、transformed等操作。

在GraphX里，没必要（从某些特定顶点开始）“遍历”全图以得到想要的边和顶点。例如，对顶点属性数据进行转换可以一下子完成，而在其他图处理系统和图数据库里，类似的操作就没那么方便了，需要两步：先执行必要的查询，然后再对查询的结果顶点集执行转换操作。

可以用给定的边RDD和顶点RDD构建一个图。一旦构建好图，就可以用函数`edges()`和`vertices()`来访问边和顶点的集合。

由于`Graph`定义的是一个属性图（正如第3章所描述），每一条边和每一个顶点都有用户自定义类表示的属性。

如图4.2的UML图所表示的，VD和ED表示用户自定义的这些类。对应的图就是参数化类型的泛型类`Graph[VD,ED]`。例如，当有一个这样的图，顶点用于表示城市和人口数量，顶点间通过道路连接起来，那么就可以用`Graph[Long,Double]`来表示这个图，顶点的数据属性是`Long`类型，表示人口数量，边的数据属性则是`Double`类型，表示城市间的距离。

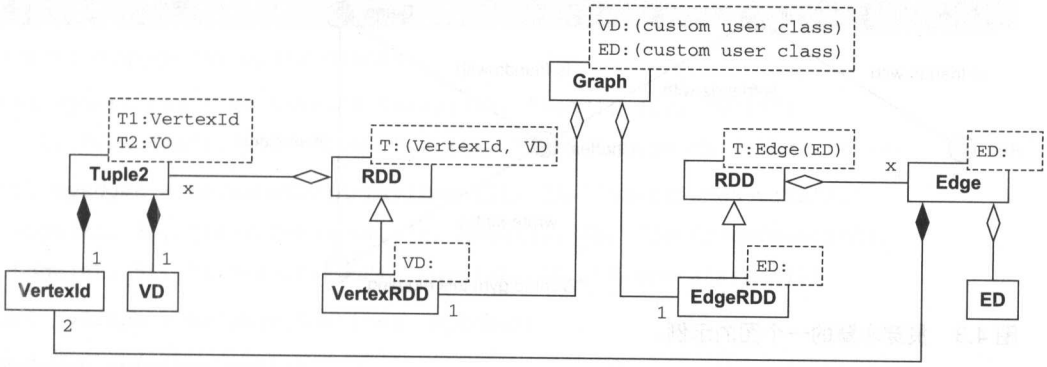
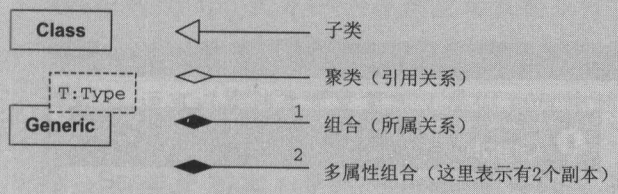


图 4.2 GraphX 里 Graph 类及其依赖的 UML 图。请注意，GraphX 将 VertexId 定义为 64 字节的 Long 类型。

UML 备忘录

如果忘了这些 UML 图标的含义，下面是对图 4.2 的补充说明：



即使是 Int 或 String 类型，也必须要指定 VD 和 ED 的具体类型。这可认为是 GraphX 的一个小的限制：不支持“缺少属性”的图，即图必须要有顶点和边属性。如果在实际场景中，你的图不需要顶点属性和边属性，可以进行最简单的处理，例如，将 Int 作为参数的类型，并为每条边和每个顶点设置相同的默认值。

GraphX 将 VertexId 定义为 64 位的 Long 类型，实际上在这个问题上也没有其他选择。注意，Edge 持有 VertexId 值，而不是顶点的引用（即 Scala Tuple2 二元组（VertexId,VD））。这是因为图在集群中是分布式存储的，不属于单个 JVM，因此一条边的顶点有可能在集群的不同节点上。

要构建一个图，可以调用这个看起来像构造函数的 Graph()。清单 4.1 中的代码构建了第 1 章的一个图，在本章其他例子中将继续使用这个图（参见图 4.3）。

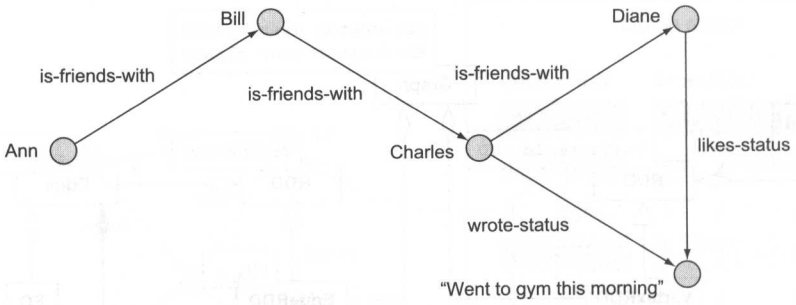
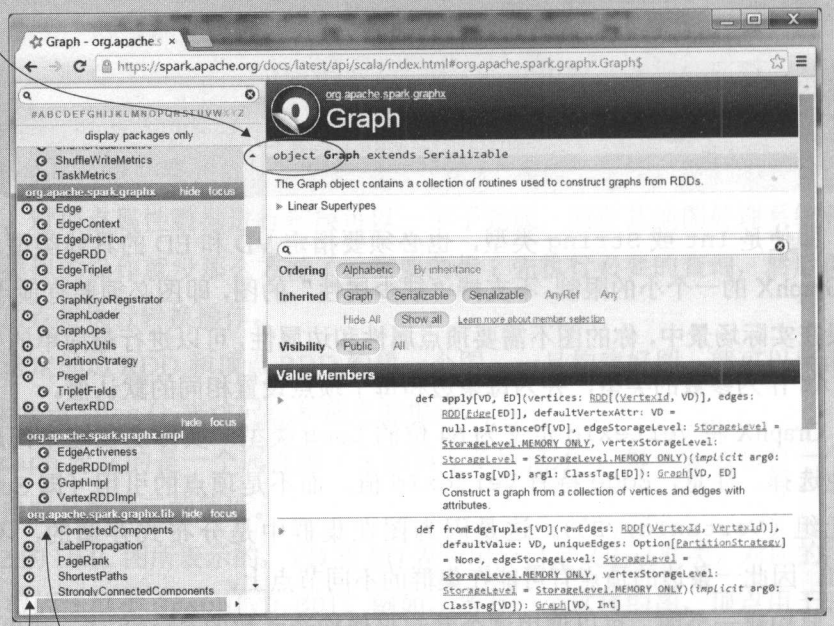


图 4.3 贯穿本章的一个图的示例。

Scala 文档中的另外一部分：伴生对象的文档

不要漏掉文档的另一部分。如 Graph 类和 Graph 伴生对象各有自己的 API。

这里是Graph伴生对象的API，与之相对的是Graph类的API。



清单 4.1 构建图 4.3 中的示例图

```
import org.apache.spark.graphx._
val myVertices = sc.makeRDD(Array((1L, "Ann"), (2L, "Bill"),
 (3L, "Charles"), (4L, "Diane"), (5L, "Went to gym this morning")))
val myEdges = sc.makeRDD(Array(Edge(1L, 2L, "is-friends-with"),
 Edge(2L, 3L, "is-friends-with"), Edge(3L, 4L, "is-friends-with"),
 Edge(4L, 5L, "Likes-status"), Edge(3L, 5L, "Wrote-status")))
val myGraph = Graph(myVertices, myEdges)
myGraph.vertices.collect
res1: Array[(org.apache.spark.graphx.VertexId, String)] = Array((4,Diane),
(1,Ann), (3,Charles), (5,Went to gym this morning), (2,Bill))
```

Scala 小贴士：用 Scala 关键词 object（而不是 class）可定义一个单例对象。这样的一个与类同名的单例对象，被称为伴生对象。GraphX API 中的 Graph 就是一个有伴生对象的例子，类和伴生对象有各自的 API。一个伴生对象，除了可以定义 apply() 方法外（例如，实现了工厂模式），还可以在伴生对象里定义类似 Java 中的静态方法。

Scala 小贴士：当一个 Scala 的类或对象中定义了函数 apply() 时，在调用 apply() 时可以省略 apply，即 Graph.apply() 简写为 Graph()。所以 Graph() 看起来像是一个构造函数，但实际上它是在调用 apply() 函数。这就是用 Scala 的 apply() 函数来实现工厂模式的一个示例，更多细节请参考 Gamma 等人的书籍 *Design Patterns*（Addison-Wesley 出版社，1994）。

Spark 小贴士：在 GraphX 的教程中你经常会看到 parallelize() 函数，而 makeRDD() 比较少见，其实二者功能一样。makeRDD() 是作者 Michael Malak 的个人偏好用法，因为他觉得这个名称更加具体且描述清晰。

清单 4.1 构建了图 4.3 所示的图。要获得相应的边集合，可参考清单 4.2 中的代码。

清单 4.2 从构建的图中获得边集合

```
myGraph.edges.collect
```

```
res2: Array[org.apache.spark.graphx.Edge[String]] = Array(Edge(1,2,
is-friends-with), Edge(2,3,is-friends-with), Edge(3,4,is-friends-with),
     Edge(3,5,Wrote-status), Edge(4,5,Likes-status))
```

由于普通RDD是无序的，顶点RDD和边RDD也就不能保证有序。

可以使用triplets()方法，根据VertexId将顶点和边联合在一起。Graph本来是将数据分开存储在对应的边RDD和顶点RDD内，triplets()函数只不过是方便地将它们联合在一起，如清单4.3所示。

清单4.3 获得triplets形式的图数据

```
myGraph.triplets.collect
```

```
res3: Array[org.apache.spark.graphx.EdgeTriplet[String,String]] =
Array(((1,Ann),(2,Bill),is-friends-with),
((2,Bill),(3,Charles),is-friends-with),
((3,Charles),(4,Diane),is-friends-with),
((3,Charles),(5,Went to gym this morning),Wrote-status),
((4,Diane),(5,Went to gym this morning),Likes-status))
```

函数triplets()返回EdgeTriplet[VD,ED]类型的RDD，它是Edge[ED]的子类，并包含边的源顶点和目标顶点的引用。如图4.4所示，EdgeTriplet类提供了访问边（以及边属性数据）以及源顶点和目标顶点属性数据的方法。接下来你会看到，GraphX可以便捷地访问边和顶点数据，让图处理任务更简单易用。

表4.1列出了EdgeTriplet的常用成员属性。

表4.1 EdgeTriplet的关键字段

字段	描述
Attr	边的属性数据
srcId	边的源顶点的ID
srcAttr	边的源顶点的属性数据
dstId	边的目标顶点的ID
dstAttr	边的目标顶点的属性数据

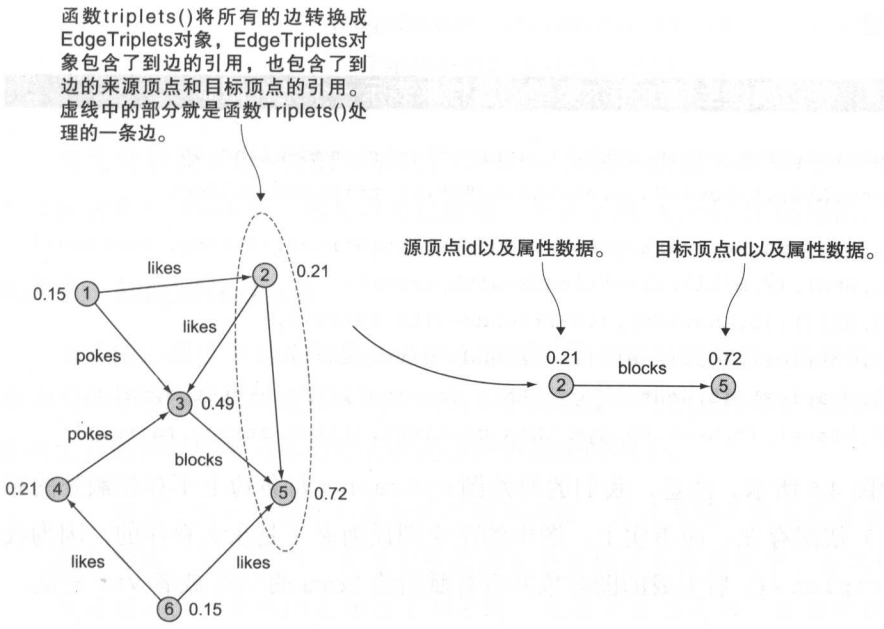


图 4.4 函数 triplets() 可以方便地对边和顶点的属性进行访问。

4.2 mapping操作

GraphX Map/Reduce 操作中最有价值的函数是 `aggregateMessages()`（代替已被废弃的 `mapReduceTriplets()` 函数）。在实际使用这个函数前，首先让我们来看看更简单的 `mapTriplets()` 函数，这也会引出 GraphX 的另一个重要概念。本书中涉及的很多转换操作都会从原来的图生成一个新图。虽然我们也可以自己对边和顶点转换来创建一个新图，这两者的最终结果或许是一致的，但这样就不能利用 GraphX 提供的底层优化功能了。

4.2.1 简单的图转换

对于在上一小节构建好的图，我们来对每一条满足这两个条件的边增加属性值：1. 属性中包含“is-friends-with”的边；2. 关系的源顶点属性中包含字母 a。那么怎么来添加属性呢？参见清单 4.4 中的代码，将 `Edge` 的类型从 `String` 转换成二元组 `(String, Boolean)`。`EdgeTriplet` 类正好适合这样的场景，即同时访问边属性

和源顶点属性。

清单4.4 在边属性上增加布尔类型的属性来表示一个条件限制

```
myGraph.mapTriplets(t => (t.attr, t.attr=="is-friends-with" &&
        t.srcAttr.toLowerCase.contains("a"))).triplets.collect

res4: Array[org.apache.spark.graphx.EdgeTriplet[String,(String, Boolean)]]
= Array(((1,Ann),(2,Bill),(is-friends-with,true)),
        ((2,Bill),(3,Charles),(is-friends-with,false)),
        ((3,Charles),(4,Diane),(is-friends-with,true)),
        ((3,Charles),(5,Went to gym this morning),(Wrote-status,false)),
        ((4,Diane),(5,Went to gym this morning),(Likes-status,false)))
```

结果如图4.5所示。注意，我们的原始图myGraph（每条边上不存在额外的布尔类型属性）依然存在。而事实上，图中的结果图反而并不是永久存在的，因为我们在调用triplets()后生成的图对象并没有赋值给Scala的val或者var变量。

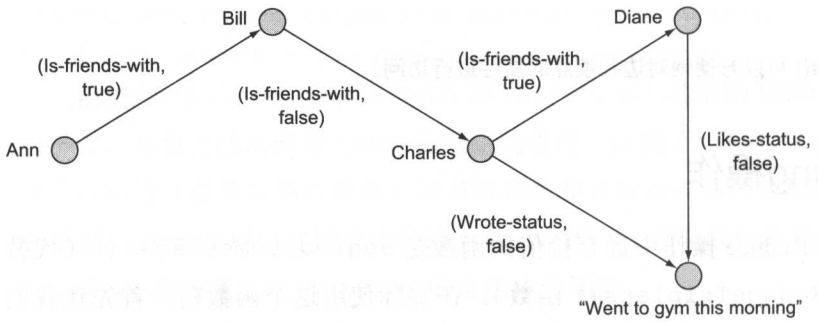


图4.5 不同于图4.3，我们使用了mapTriplets()函数来对边的类型进行转换。图4.3中边的类型是String，而这里边的类型是二元组(String,Boolean)。

尽管mapTriplets()有两个可选参数，但这里我们只用了第一个参数。这个参数是一个匿名函数，它传入一个EdgeTriplet对象作为输入参数，返回一个包含二元组(String,Boolean)的Edge类型。

> **Scala小贴士**：如果没有声明匿名函数的返回类型，Scala会根据函数中返回的结果推断出返回类型。在上述例子中，如果我们希望Scala编译器双重检查类型（以及代码标注的类型），应该按以下的方式写代码：
>
> ```
> myGraph.mapTriplets((t => (t.attr, t.attr=="is-friends-with" &&
>                         t.srcAttr.toLowerCase.contains("a"))) :
> ```

```
                (EdgeTriplet[String,String] =>
                  Tuple2[String,Boolean]) )
    .triplets.collect
```

与允许转变 Edge 类的 mapTriplets() 类似，函数 mapVertices() 允许我们直接转变 Vertex 类。你可以自己探索一下 mapVertices() 的用法。

4.2.2 Map/Reduce

很多图处理任务需要聚集从周围本地邻居顶点发出的消息。这里的邻居指的是顶点周围直接相关联的边和顶点。在第 5 章会涉及一些如三角形计数的经典图算法，那时会看到如何将来自本地邻居的消息聚集起来的例子。

> **定义**：当一个顶点和另外两个顶点相连，并且另两个顶点也是相连的，就会形成一个三角形。在用于展示朋友关系的社交媒体图中，当其他人是我的朋友，并且他们也相互是朋友时，就会产生三角形。我们可以认为，一个用户涉及的三角形个数，反映了该用户周边社区的连通性。

为了识别一个顶点是否处于三角形中，我们需要考虑与该顶点相连的边集，及这些边除该顶点之外的顶点集，以及这些点之间是否也有相连的边。对于每一个顶点而言，这意味着要考虑它的邻居们的消息。

我们将三角形个数统计的实现放在第 5 章，这里仅考虑一个简单的例子，重点理解在处理和聚集邻居顶点消息过程中的核心概念。这个思路和经典的 Map/Reduce 范例（参考第 1 章）很相似，与 Map/Reduce 操作一样，我们会定义转换函数（map）用于处理邻居顶点的独立结构；然后，经过转换后的输出结果会被合并，用于更新顶点的消息（reduce）。

我们的例子会统计每个顶点的“出度”——即对于每个顶点而言，离开该顶点的边的条数。为了完成上述操作，我们会间接处理每条边以及与边相关联的源顶点和目标顶点。相比于直接统计从每个顶点出发的边条数，我们会让每条边发出消息到关联的源顶点。这两种方法是等效的。汇总这些消息后，就可以得到我们想要的答案。

清单 4.5 中的代码使用了 aggregateMessages() 函数，这也是在 GraphX 中完成这个任务唯一需要的函数。

清单 4.5 使用 aggregateMessages[]() 计算每个顶点的出度

```
myGraph.aggregateMessages[Int](_.sendToSrc(1), _ + _).collect
res5: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((4,1), (1,1),
(3,2), (2,1))
```

返回的数组里包含了记录顶点 ID 和顶点出度值的二元组。顶点 #4 只有一条出边，而顶点 #3 拥有两条出边。

这是怎么实现的呢？为了理解这段代码的原理，我们将它拆分成几个部分。首先是 aggregateMessages 的函数签名：

```
def aggregateMessages[Msg](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg)
    : VertexRDD[Msg]
```

首先应该注意的是这个函数的参数化类型：Msg（message 的缩写，接下来会进行说明）。Msg 类型表示函数返回结果数据的类型，我们想要对从顶点传出的边数进行统计，Msg 的具体类型选择 Int 比较合适。

> **Scala 小贴士**：由于 Scala 编译器可以进行类型推断，你可能会希望 aggregateMessages 也能根据第一个参数（匿名函数）的返回类型自动推断出最终返回结果的类型。毕竟，它们在 aggregateMessages 中都被声明为 Msg 类型。这种情况下 Scala 编译器并不能推断出结果类型，因为 Scala 是从左到右处理的编译器，而匿名函数在代码里的位置要比调用的函数名更靠后一点。

aggregateMessages 的两个参数是 sendMsg 和 mergeMsg，提供了转换和聚合的能力。

sendMsg

sendMsg 函数以 EdgeContext 作为输入参数，没有返回值（之前章节也提到过，Scala 的 Unit 等价于 Java 的 void）。EdgeContext 和 EdgeTriplet 都是参数化类型的类（type-parameterized class）。EdgeContext 接口的实现类和 EdgeTriplet 有两个相同的成员变量，而 EdgeContext 多提供了两个消息的发送函数。

- sendToSrc：将 Msg 类型的消息发送给源顶点。
- sendToDst：将 Msg 类型的消息发送给目标顶点。

这两个方法是 aggregateMessages 工作原理的重要组成部分。传递的消息其实就是发送给顶点的一些数据。对于图中的每条边，我们可以选择向源顶点或目标顶点（或同时向这两个顶点）发送消息。在 sendMsg 方法内部，参数 EdgeContext 用于检查边、源顶点、目标顶点三者的属性值。在这个例子中，我们需要计算每个顶点发出的边数，所以在边上将包含整数 1 的消息发送到源顶点。

mergeMsg

每个顶点收到的所有消息都会被聚集起来传递给 mergeMsg 函数。这个函数定义了如何将顶点收到的所有消息转换成我们需要的结果。在示例代码中，我们将所有发给源顶点的数字 1 累加起来得出边的总数。这就是匿名函数 _ 使用 + 操作完成的。

在每个顶点上应用 mergeMsg 函数最终返回一个 VertexRDD[Int] 对象。VertexRDD 是一个包含了二元组的 RDD，包括了顶点的 ID 以及该顶点的 mergeMsg 操作的结果。需要注意的一点是，由于顶点 #5 不含有任何出边，它接收不到任何消息，所以它不会出现在结果 VertexRDD 中。

清理结果

这些原始的 VertexId 不容易理解，所以我们为顶点添加上可读的名字。在清单 4.6 中，我们使用到了 Spark RDD 的 join() 操作，这是 PairRDDFunctions 的一个方法。

清单 4.6 RDD 的 join() 操作用于匹配 VertexId 与顶点数据

```
myGraph.aggregateMessages[Int](_.sendToSrc(1),
 _ + _).join(myGraph.vertices).collect

res6: Array[(org.apache.spark.graphx.VertexId, (Int, String))] =
Array((4,(1,Diane)), (1,(1,Ann)), (3,(2,Charles)), (2,(1,Bill))
```

Spark 小贴士：当你在任何时候遇到一个含二元组的 RDD 时，Spark 都会在需要时自动把 RDD[] 转换成 PairRDDFunctions[]，其中二元组是键值对的形式。join() 是众多可用于二元组形式 RDD[Tuple2] 转换的函数之一。在 REPL 环境中这样的转换是自动进行的，而在一个已编译的 Scala 程序里，你需要额外导入 import org.apache.spark.SparkContext._（除了普通的 import org.apache.spark.SparkContext 外）。

当然，上面的做法看起来还是有点烦琐。其实后面用不到这些VertexId了，所以我们可以使用RDD的map()函数去掉它们，可以使用二元组的swap()方法交换两个元素的顺序，将可读的顶点名放在出度值之前使得输出更为美观，如清单4.7所示。

清单4.7 使用map()和swap()整理输出

```
myGraph.aggregateMessages[Int](_.sendToSrc(1),
 _ + _).join(myGraph.vertices).map(_._2.swap).collect

res7: Array[(String, Int)] = Array((Diane,1), (Ann,1), (Charles,2),
(Bill,1))
```

现在进行最后的整理，我们怎么重新获得丢弃的#5顶点呢？可以使用rightOuterJoin()来代替join()，如清单4.8所示。

清单4.8 使用rightOuterJoin()代替join()找回丢失的顶点

```
myGraph.aggregateMessages[Int](_.sendToSrc(1),
 _ + _).rightOuterJoin(myGraph.vertices).map(_._2.swap).collect

res8: Array[(String, Option[Int])] = Array((Diane,Some(1)), (Ann,Some(1)),
 (Charles,Some(2)), (Went to gym this morning,None), (Bill,Some(1)))
```

啊！新出现的Some和None是怎么回事呢？这是因为当在连接表中不存在相关记录时，外连接会产生值为null或空的字段。这也是Scala为了避免null问题的解决办法（尽管它在你需要的时候依然可以提供null值，并且在和Java代码对接时这常常是不可避免的）。Some和None来自Scala的Option[]类，去掉这些值需要使用Option[]的getOrElse()方法。接下来，为了更好地处理二元组，我们不再简单地使用swap()处理，如清单4.9所示。

清单4.9 Option[]的getOrElse()方法可用于整理rightOuterJoin()的输出

```
myGraph.aggregateMessages[Int](_.sendToSrc(1),
 _ + _).rightOuterJoin(myGraph.vertices).map(
 x => (x._2._2, x._2._1.getOrElse(0))).collect

res9: Array[(String, Int)] = Array((Diane,1), (Ann,1), (Charles,2), (Went to
     gym this morning,0), (Bill,1))
```

Scala 小贴士：Scala 使用 Option[] 来代替 null，这为函数式编程提供了更好的便利，Option[] 可被看作是一个最小集合（包含零个或者一个元素）。函数式编程中的 flatmap() 函数、for 推导式和偏函数等不能处理 null，但可以处理 Option[]。

4.2.3 迭代的 Map/Reduce

大多数的算法都包含多次迭代。aggregateMessages 可用于这类算法，其仅需要基于邻边和顶点发送过来消息来不断更新每个顶点的状态。

为了实现上述过程，我们实现了在图中寻找与顶点距离最远的根顶点的算法。算法的最终结果，我们希望为每个顶点标记上离它最远的根顶点的距离。

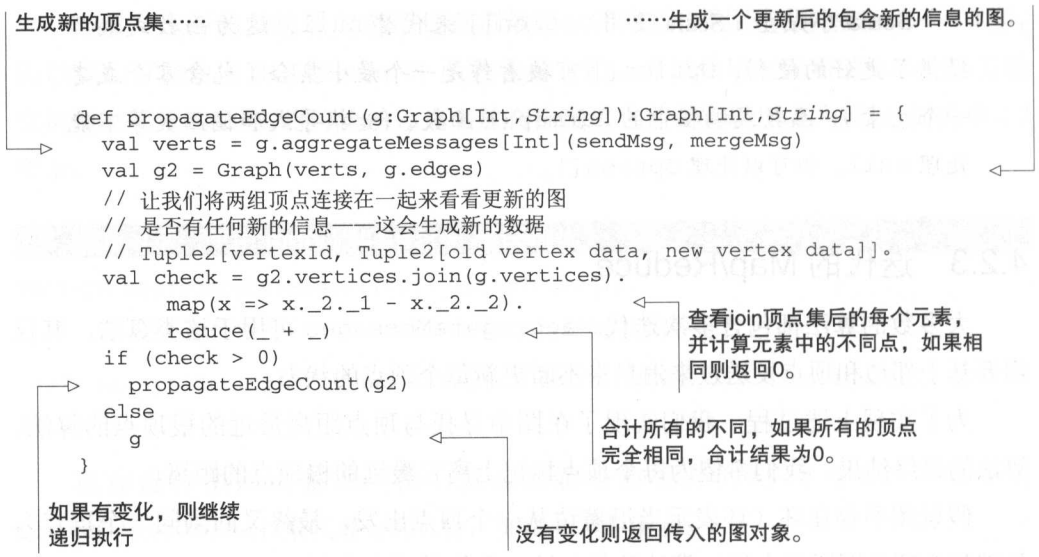
假设图不存在环（环表示当沿着边从一个顶点出发，最终又回到同一个顶点）。处理包含环的图通常会增加算法的复杂性，我们稍后会介绍一些处理环的策略。

首先我们对 aggregateMessages 会调用到的 sendMsg 和 mergeMsg 进行定义。我们不把 sendMsg 和 mergeMsg 作为匿名函数传给 aggregateMessages 函数，而是显式地定义 sendMsg 和 mergeMsg 函数，这样会使得代码更为清晰。

在函数式编程中实现迭代通常采用递归的方式，所以接下来我们会定义一个用于递归的辅助函数 propagateEdgeCount，它会持续调用 aggregateMessages（如清单 4.10 所示）。

清单 4.10 迭代（通过递归的方式）的 Map/Reduce，用于寻找距离最远的顶点

```
// sendMsg 函数作为参数传入 aggregateMessages。
// 这个函数会在图中的每条边上被调用。这里 sendMsg 只是简单的累加计数器。
def sendMsg(ec: EdgeContext[Int,String,Int]): Unit = {
  ec.sendToDst(ec.srcAttr+1)
}

// 这里我们定义了 mergeMsg 函数，这个函数会在所有的消息传递到顶点后被重复调用。
// 消息经过合并后，最终得出结果为包含最大距离值的顶点。
def mergeMsg(a: Int, b: Int): Int = {
  math.max(a,b)
}
```



在上述代码中，propagateEdgeCount() 会将当前距离加 1，并将它发送给每条边的目标顶点。然后目标顶点对它接收到的所有消息进行 max() 操作，并进行距离的更新。

值得注意的一个重点是，每轮迭代后，我们对原始图和更新图进行比较，来定义结束递归的条件。当这两个图没有差异时则结束迭代。这里起关键作用的是 reduce(_+_)，因为我们知道更新后的顶点间的距离至少和原来的距离值一样大，因此它们的差不可能是负数。不会出现负数与正数相加最终得到零这种特殊情况。

> **提示：**考虑到性能，应该避免使用 join()，而是应通过增加一个新的布尔类型的顶点属性的方式，这个布尔属性值表示是否有距离更新。然后我们可以对这个布尔值执行 reduce() 操作，而不需要对比上一个迭代的 VertexRDD 来作为递归退出条件。

现在我们把递归函数准备好了，将 myGraph 图作为输入，首先要进行初始化。这是任何迭代算法都需面对的关键问题：怎么开始呢？我们需要理清任务的需求，开始的时候每个顶点已知的消息是什么，我们期待得到什么样的结果。我们期待的是一个在图中经过了最远距离的整数。在开始的时候我们不知道距离值，所以将每个顶点的值设置为 0，让算法逐步在图中传播消息：

```
val initialGraph = myGraph.mapVertices((_,_) => 0)
```

```
propagateEdgeCount(initialGraph).vertices.collect

res10: Array[(org.apache.spark.graphx.VertexId, Int)] = Array( (1,0),
    (2,1)), (3,2), (4,3), (5,4) )
```

#5 顶点与根顶点的距离最远，距离值为 4。

你已经可以看到，实现一个迭代算法确实很方便。关键点在于要考虑信息（“消息”）是如何通过边来传播和累计，以便多次迭代后得到我们想要的结果的。

4.3 序列化/反序列化

在第 2 章中我们介绍了如何使用 GraphX 的 `GraphLoader` API 来读取边信息。但那份数据既没有包含顶点的属性，也没有包含边的属性，而 GraphX 的一个突出优点就是它可以处理属性图。这里我们提供了一些定制化的代码，用于读写二进制和 JSON 格式的属性图。至于 RDF 格式的数据，即标准的三元组图数据（不是属性图），会在第 8 章进行详细介绍。

4.3.1 读 / 写二进制格式的数据

在本小节，我们会读写标准的 Hadoop 序列文件，即包含一系列序列化对象的二进制文件。Spark RDD 的 API 函数 `saveAsObjectFile()` 默认用标准的 Java 序列化来序列化顶点和边对象，将数据保存为 Hadoop 序列文件，代码如清单 4.11 所示。

清单 4.11 读写持久化文件

```
myGraph.vertices.saveAsObjectFile("myGraphVertices")
myGraph.edges.saveAsObjectFile("myGraphEdges")
val myGraph2 = Graph(
    sc.objectFile[Tuple2[VertexId,String]]("myGraphVertices"),
    sc.objectFile[Edge[String]]("myGraphEdges"))
```

保存到HDFS中，文件路径为“hdfs://localhost:8020/myGraphVertices”。

Spark 小贴士：与名字描述不同，`saveAsObjectFile()` 函数会在一个目录中保存多个文件，一个 Spark 分区对应其中一个文件。传入参数作为目录名。

这个简单的图对象用 `String` 作为顶点和边的属性类，但由于前面使用到了 Java 的序列化方法，事实上它可以处理任意复杂的类，这些复杂类都可以作为顶点

和边的属性。用实际的类名来替代 objectFile[]() 类型参数中的 String 即可。

前面的代码没有指定采用的文件系统，所以它会保存到本地文件系统中。在实际的 Spark 集群中，需要预先考虑为目标目录名加上合适的 URL 前缀。例如，hdfs://localhost:8020/myGraphVertices。

GraphX 对象序列化

在 Spark 核心代码里，所有的对象都需要是可序列化的，以便把序列化的对象传输到 worker 节点上。Spark 默认采用的是 Java 的序列化，但 Spark 也集成了优秀的 Kryo，这是一个更高效的序列化框架。在 Spark 1.0 之前，Spark 中的 Kryo 存在很多问题，到了 Spark 1.6 版本，还存在许多与 Kryo 相关的 Jira。第三个可选的序列化方案是使用 Java 的 Externalizable，来实现自定义的序列化策略——更贴切地说，使用第三方库，例如 Avro 和 Pickling——但这种方法并不那么简单易用。幸运的是，Spark 的 saveAsObjectFile() 提供了另外一个优化策略：压缩。当对象经过序列化后，它们一般还会经过编码解码器。尽管序列化不会压缩，但 saveAsObjectFile() 会在序列化后对数据进行压缩。

保存到一个文件中：小技巧

使用 saveAsObjectFile() 保存数据到 HDFS 或 S3 时，为了避免保存到多个不同的文件中，可以先执行 coalesce(1,true)，代码如下：

```
myGraph.vertices.coalesce(1,true).saveAsObjectFile("myGraphVertices")
```

这个小技巧的缺点是，整个 RDD 要保存到一个分区中，这意味着所有数据需要保存到一个执行器中。这在处理小图时有用，但不可用于生产环境的大图。

保存到一个文件中：正确的做法

假设你在 HDFS 中进行操作，产生单一文件的正确做法是首先允许 Spark 创建多个小文件（Spark 会并行化执行该操作），然后将它们合并到单一的文件中。有两种方式可以完成后一步的操作：用命令行或者用 Hadoop Java API。

在下面的示例中，首先确保我们将数据保存成 HDFS 文件而不是本地文件。在用 Cloudera Quickstart VM 时可使用以下的 URL，这与具体的环境有关：

```
myGraph.vertices.saveAsObjectFile(
  "hdfs://localhost:8020/user/cloudera/myGraphVertices")
```

为了合并 myGraphVertices 文件夹中的多个文件，一个可选的操作是使用命令行，它会读取 HDFS 的多个文件并在本地文件系统中创建一个单一文件：

```
hadoop fs -getmerge /user/cloudera/myGraphVertices myGraphVerticesFile
```

和 getmerge 命令等价的 API 函数是 copyMerge()，它可将最终文件保存在 HDFS 中，代码如清单 4.12 所示。

清单 4.12 用 Hadoop Java API 保存 HDFS 文件为单一文件

```
import org.apache.hadoop.fs.{FileSystem, FileUtil, Path}
val conf = new org.apache.hadoop.conf.Configuration
conf.set("fs.defaultFS", "hdfs://localhost")
val fs = FileSystem.get(conf)
FileUtil.copyMerge(fs, new Path("/user/cloudera/myGraphVertices/"),
  fs, new Path("/user/cloudera/myGraphVerticesFile"), false, conf, null)
```

Scala 小贴士：你可以使用花括号在同一行内合并同一个 package 的多个不同类的导入。

现在你仍然要处理两个不同的文件：一个是由上述代码产生的顶点文件，另一个是边的文件（创建方法类似）。为了真正地用单一的文件进行管理，可以选择用 graph.triplets.saveAsObjectFile() 来保存三元组。需要注意，这样会产生额外的磁盘浪费，因为每条边关联的顶点会有重复，即同一个顶点会被用于构成多条边，那么边数据中就会有重复的顶点。在这个例子中，将三元组存储成单一文件的方式所占用的磁盘空间，大概是分开存储边和顶点方式的 20 倍。

如果你打算使用 S3 代替 HDFS，命令行 hadoop fs -getMerge 和 Hadoop API 的 copyMerge() 都不能用。网络上有大量的 Shell 脚本和图形界面工具可用于在 S3 中完成上述操作。

4.3.2 JSON 格式

如果你倾向于序列化后保存成可读性好的文件格式，可以用 JSON 的库。在 Scala 处理 JSON 方面，对 Java 中著名的 Jackson 库进行封装会是一个不错的选择。

Jerkson包装器一直被广泛使用，直到2012年被废弃。在这方面，Jackson甚至还发布了jackson-module-scala。它的语法虽然不像其他Scala库一样简明，但它能正常使用。尽管Jackson已经可以在Spark REPL中使用，但仍需执行清单4.13中的命令来开启jackson-module-scala功能。

清单4.13 用于在REPL中启用jackson-module-scala的命令行

```
wget http://repo1.maven.org/maven2/com/fasterxml/jackson/module/
➥ jackson-module-scala_2.10/2.4.4/jackson-module-scala_2.10-2.4.4.jar wget
http://repo1.maven.org/maven2/com/google/guava/guava/14.0.1/
➥ guava-14.0.1.jar
./spark-shell --jars jackson-module-scala_2.10-2.4.4.jar,guava-14.0.1.jar
```

请注意上述给定的版本号（Scala 2.10、Jackson 2.4.4以及相关联的Guava 14.0.1）是针对Spark 1.6版本的。如果你使用的是其他版本的Spark，请找到启动REPL时需要的Scala版本，以及Maven的pom.xml中用到的Jackson和Guava版本。在Spark 1.6中，所有依赖到的jar包的版本号都集中在Spark的根pom.xml文件中。jackson-module-scala的版本号需要和Spark内置的Jackson的版本号保持一致。

首先我们来看看Jackson是如何利用我们已看到的Spark特性将图序列化为JSON格式的。然后我们会介绍一个新的特性，mapPartitions，它在某些情况下可以带来显著的性能提升。

一旦REPL启动成功，就可以参考清单4.1来再次创建myGraph对象，然后尝试将它序列化为JSON格式，代码如清单4.14所示。

清单4.14 序列化为JSON的简单方法

```
myGraph.vertices.map(x => {
    val mapper = new com.fasterxml.jackson.databind.ObjectMapper()
    mapper.registerModule(
        com.fasterxml.jackson.module.scala.DefaultScalaModule)
    val writer = new java.io.StringWriter()
    mapper.writeValue(writer, x)
    writer.toString
}).coalesce(1,true).saveAsTextFile("myGraphVertices")
```

注意：输出文件本身并不是JSON可解析的；然而，该文件的每一行内容都是有效的JSON格式。相比在除了最后一行外的每一行的后面添加

逗号并且在文件开始和末尾添加开闭方括号，这样的方式更有助于分布式存储和分布式处理。

对于每一个顶点，我们都会构建一个全新的 JSON 解析器。为应对这种情况，Spark API 提供了 mapPartitions() 作为 map() 的替代方案。清单 4.15 展示了性能更佳的处理方法。

清单 4.15 性能更佳的 JSON 序列化 / 反序列方式

```
import com.fasterxml.jackson.core.`type`.TypeReference
import com.fasterxml.jackson.module.scala.DefaultScalaModule

myGraph.vertices.map(x => {
    val mapper = new com.fasterxml.jackson.databind.ObjectMapper()
    mapper.registerModule(
        com.fasterxml.jackson.module.scala.DefaultScalaModule)
    val writer = new java.io.StringWriter()
    mapper.writeValue(writer, x)
    writer.toString
}).coalesce(1,true).saveAsTextFile("myGraphVertices")

myGraph.vertices.mapPartitions(vertices => {
    val mapper = new com.fasterxml.jackson.databind.ObjectMapper()
    mapper.registerModule(DefaultScalaModule)
    val writer = new java.io.StringWriter()
    vertices.map(v => {writer.getBuffer.setLength(0)
                       mapper.writeValue(writer, v)
                       writer.toString})
}).coalesce(1,true).saveAsTextFile("myGraphVertices")

myGraph.edges.mapPartitions(edges => {
    val mapper = new com.fasterxml.jackson.databind.ObjectMapper();
    mapper.registerModule(DefaultScalaModule)
    val writer = new java.io.StringWriter()
    edges.map(e => {writer.getBuffer.setLength(0)
                    mapper.writeValue(writer, e)
                    writer.toString})
}).coalesce(1,true).saveAsTextFile("myGraphEdges")
```

```
val myGraph2 = Graph(
    sc.textFile("myGraphVertices").mapPartitions(vertices => {
        val mapper = new com.fasterxml.jackson.databind.ObjectMapper()
        mapper.registerModule(DefaultScalaModule)
        vertices.map(v => {
            val r = mapper.readValue[Tuple2[Integer,String]](v, new
            TypeReference[Tuple2[Integer,String]]{})
            (r._1.toLong, r._2)
        })
    }),
    sc.textFile("myGraphEdges").mapPartitions(edges => {
          val mapper = new com.fasterxml.jackson.databind.ObjectMapper()
          mapper.registerModule(DefaultScalaModule)
          edges.map(e => mapper.readValue[Edge[String]](e,
              new TypeReference[Edge[String]]{}))
    })
)
```

Spark小贴士：当不是为各个RDD元素依次都执行初始化，而是需要一次性完成大量RDD元素的初始化工作时，并且包括从第三方类库创建的初始化的对象都可以序列化（对象可序列化，Spark才能够把对象序列化后的二进制数据传输到集群的多个worker节点上），那么可以用mapPartitions()代替map()。

Scala小贴士：如果你需要用Scala保留的关键字，可以在它的两边加上重音符（`）。在Scala中调用Java的库时会经常用到这个技巧。

请注意，由于Jackson本身的缺陷，我们需要读取整数类型的顶点ID，然后把整型转换为Long类型。这把顶点ID最大取值范围限制在了20亿以内。并且Jackson让我们在readValue[]()里重复了两次顶点和边的属性类泛型参数（在本例中为String），所以如果你需要使用一个定制化的类，要记得在所有地方进行替换。最后，和二进制的例子相似，当使用到分布式文件系统时，要记得去掉coalesce(1,true)。

4.3.3 Gephi 可视化软件的 GEXF 格式

Gephi 是一个强大的开源的图可视化软件，GEXF 是它原生的 XML 格式。在本小节中，正如接下来的清单 4.16 和图 4.6 所示，我们把下面的示例导出并序列化为 .gexf 格式的文件。Gephi 是开源的，并可在 gephi.github.io 下载到，它的用法在附录 B 中进行了说明。

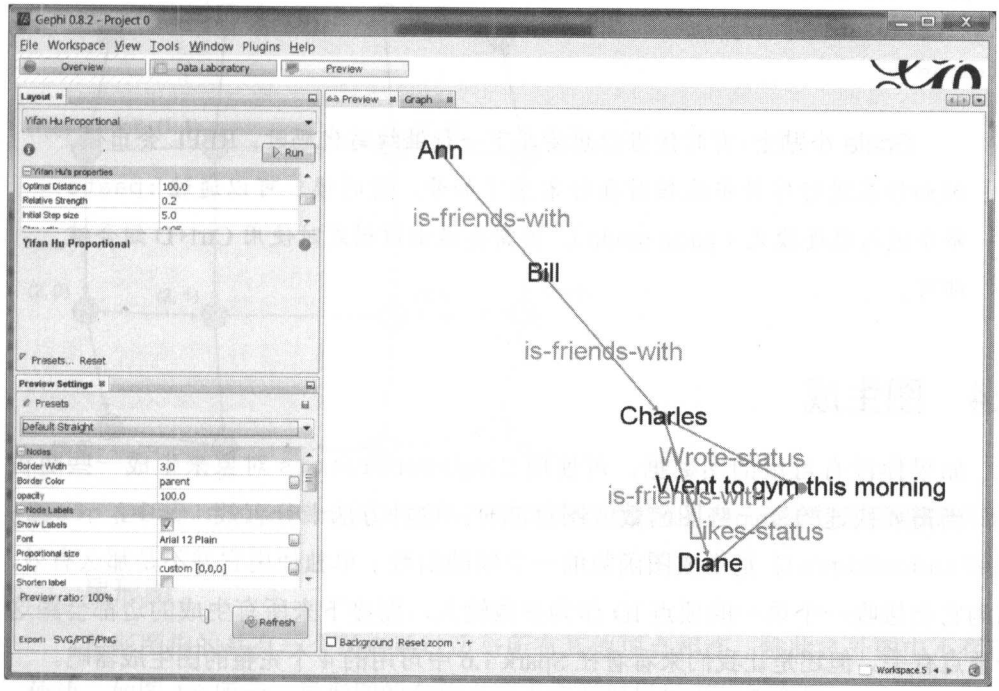


图 4.6 使用 Gephi 加载生成的 .gexf 文件。

清单 4.16 导出 Gephi 软件所支持的 GEXF 格式的文件

```
def toGexf[VD,ED](g:Graph[VD,ED]) =
    "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
    "<gexf xmlns=\"http://www.gexf.net/1.2draft\" version=\"1.2\">\n" +
    "  <graph mode=\"static\" defaultedgetype=\"directed\">\n" +
    "    <nodes>\n" +
    g.vertices.map(v => "      <node id=\"" + v._1 + "\" label=\"" +
                        v._2 + "\" />\n").collect.mkString +
    "    </nodes>\n" +
    "    <edges>\n" +
```

```
    g.edges.map(e => "      <edge source=\"" + e.srcId +
                     "\" target=\"" + e.dstId + "\" label=\"" + e.attr +
                     "\" />\n").collect.mkString +
    "    </edges>\n" +
    "  </graph>\n" +
    "</gexf>"
val pw = new java.io.PrintWriter("myGraph.gexf")
pw.write(toGexf(myGraph))
pw.close
```

> **Scala小贴士：**有时候当你想要在下一行继续写代码时，REPL会出错，例如你在进行字符串连接时在行末用了加号。这时候，可以通过:paste命令进入粘连模式（paste mode）。当需要退出该模式时使用Ctrl+D组合键即可。

4.4 图生成

如果你没有可用的图数据，可使用GraphGenerators对象来生成一些随机图。当需要快速验证一些图函数或图算法时，这种方法十分有效。其中，generateRandomEdges()是生成图函数的一个辅助函数；单独使用它并不是那么有效，因为它会接收一个单一的顶点ID作为参数输入，而接下来所有生成的边都会跟这个顶点有关。但还是让我们来看看在Spark 1.6中可用的4个完整的图生成器吧。

4.4.1 确定的图

我们首先要介绍的这两个图结构并不是随机的：网格图和星形图。接下来的清单4.17假设你已经执行了import org.apache.spark.graphx._。例如，util.GraphGenerators指的是org.apache.spark.graphx.util.GraphGenerators。

网格图

网格图的顶点和边符合特定的模式，就像是在一个二维的网格或矩阵中。每一个顶点都用它在网格中行和列的位置作为标签（例如，左上顶点的标签为(0, 0)）。每个顶点都和它上、下、左、右的直接邻居相连。清单4.17中的代码展示了如何创

建一个 4×4 的网格图。图的布局如图 4.7 所示。

清单 4.17 生成网格图

```
val pw = new java.io.PrintWriter("gridGraph.gexf")
pw.write(toGexf(util.GraphGenerators.gridGraph(sc, 4, 4)))
pw.close
```

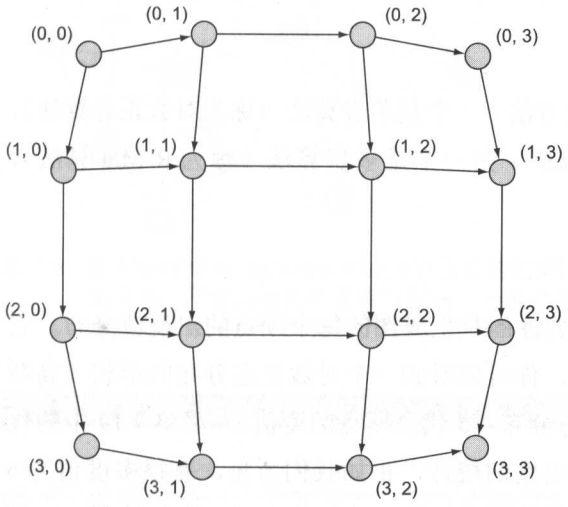


图 4.7 在 Gephi 中对 gridGraph() 生成的图进行可视化。网格图不存在随机性，并且网格是完整的。

星形图

星形图指的是有一个顶点通过边与所有其他顶点相连，除此之外图中不存在其他边。如图 4.8 所示，星形图的名称来源于它类似星星形状的布局。

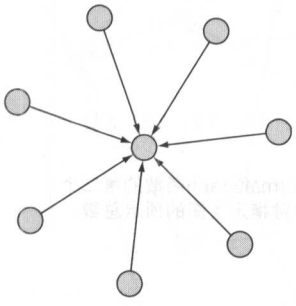


图 4.8 在 Gephi 中对 starGraph() 生成的图进行可视化。与 gridGraph() 类似，它并不是随机图。

该图是由函数 `GraphGenerators.starGraph` 生成的，通过第二个参数指定了顶点的数目。顶点 0 一直会作为星形图的中心，所以在以下清单 4.18 所示的代码中调

用 GraphGenerators.starGraph(sc,8)，会得到顶点 0 与其他 7 个顶点相连的图。

清单 4.18 生成星形图

```
val pw = new java.io.PrintWriter("starGraph.gexf")
pw.write(toGexf(util.GraphGenerators.starGraph(sc, 8)))
pw.close
```

4.4.2 随机图

GraphX 提供了两种随机生成图的方法：一个是单步算法（称为对数正态算法），它将特定数量的边与各个顶点关联起来；另一个是多步算法（称为 R-Mat 算法），它会生成与现实世界比较相近的图。

基于度的：对数正态图

对数正态图（log normal graph）关注的是生成图的每个顶点的出度值分布。它保证在对所有的出度值绘制直方图时，你可以看到一个对数正态分布的形状（高斯钟形曲线），这意味着 log(d) 服从正态分布（d 代表顶点的度值）。清单 4.19 中的代码可以和清单 4.5 中的代码一起进行出度的统计，正如我们所见，有很多度值为 6 的顶点，并且会向左边（具有更小度值的顶点）和右边（具有更大度值的顶点）呈衰减趋势。图 4.9 展示了一个可能的输出图。

清单 4.19 生成一个对数正态图

```
val logNormalGraph = util.GraphGenerators.logNormalGraph(sc, 15)
val pw = new java.io.PrintWriter("logNormalGraph.gexf")
pw.write(toGexf(logNormalGraph))
pw.close
logNormalGraph.aggregateMessages[Int](
    _.sendToSrc(1), _ + _).map(_._2).collect.sorted

Res11: Array[Int] = Array(2, 3, 4, 6, 6, 6, 6, 8, 9, 9, 9, 10, 10, 13, 14)
```

logNormalGraph函数的第二个参数指定了图的顶点总数。

基于程序化流程的：R-MAT 图

R-MAT，代表*递归矩阵*，用于模拟典型的社交网络架构。与之前的基于度值的 logNormalGraph() 函数相反，rmatGraph() 进行“程序化”的过程。它根据预先设定的每个象限的概率，每次将边添加到图的某个象限中（再到象限中的象限，

递归进行），如清单 4.20 中的代码所示，生成的图如图 4.10 所示。

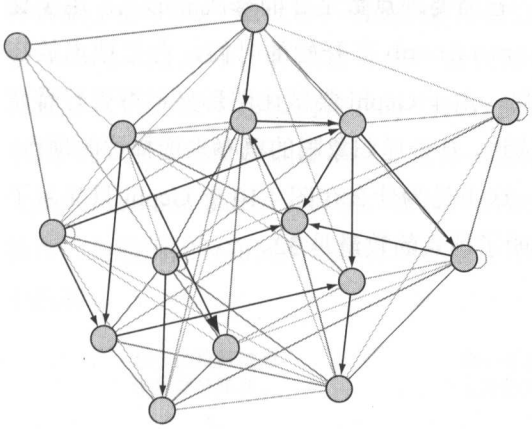


图 4.9 在 Gephi 中对 logNormalGraph() 生成的图进行可视化。这里的唯一约束是每一个顶点的出度值，不会对边的位置进行限制。一些边会拥有相同的起始点和终止点。而且一些顶点对之间会存在有多条平行的边，在 Gephi 的可视化中显示为颜色更深和箭头更大的边。

清单 4.20 生成 R-MAT 图

```
val pw = new java.io.PrintWriter("rmatGraph.gexf")
pw.write(toGexf(util.GraphGenerators.rmatGraph(sc, 32, 60)))
pw.close
```

rmatGraph函数的第二个和第三个参数是要求的顶点数和独立的边数——顶点数被取值为最近的一个2的幂值。

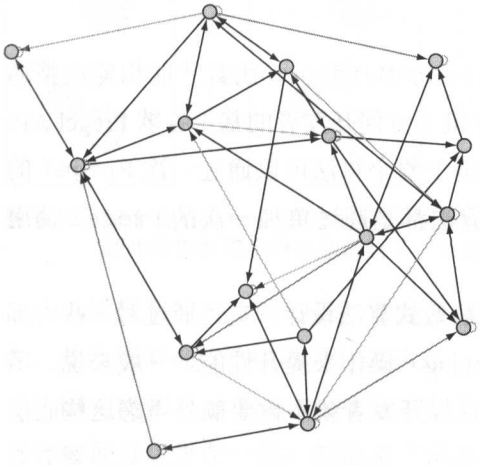


图 4.10 rmatGraph 的递归象限分割，会留下一些相对孤立的顶点，同时会让一些顶点群组高度内聚。

rmatGraph假设作为参数传递进来的顶点数是2的幂次。如果不是，rmatGraph会自动将它增大到最近的2的幂次方值。它需要顶点数是2的幂次的形式，用于递归地进行象限的分割，如图4.11所示。rmatGraph在开始的时候，会把顶点放在网格里（图4.10显示的不是网格状的布局，由于Gephi尝试让图形显示得更好看而调整了顶点的位置），然后随机地逐一生成边，基于递归象限的概率来选择边的顶点。一些顶点有可能最终没有边与它们相连，这也是为什么在图4.10中Gephi只显示了16个顶点的原因，尽管在清单4.20里指明了顶点的数量是32。

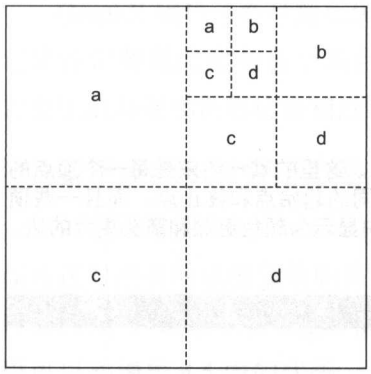


图4.11 递归象限来源于2004年Chakrabarti等人的论文R-MAT: A Recursive Model for Graph Mining。GraphX采用硬编码的方式将概率设置为a=0.45, b=0.15, c=0.15, d=0.25。

4.5 Pregel API

我们在2.3节讲过，通过多次执行aggregateMessages函数才能构建完整的算法。由于这个需求很常见，所以GraphX提供了方便开发者的基于谷歌Pregel API的迭代算法。这个API的设计相当简明，以至于整个算法可以通过一次Pregel的调用来完成。事实上，很多GraphX的固有算法都是通过单独一次的Pregel调用来实现的。

GraphX的Pregel API提供了一个简明的函数式算法设计。它还通过对一些内部数据集的缓存（caching）和释放缓存（uncaching）操作来提升性能。一般来说，需要有一定的技巧才能保证这些操作不出错，这样开发者就不需要额外考虑这些底层性能调优的细节。

在GraphX中实现Pregel，使用到了整体同步并行计算模型（Bulk Synchronous Parallel，BSP）的理念。正如它的名字所示，BSP是一个20世纪80年代提出的并

行处理模型。BSP 并不是为图处理而提出的，但当谷歌实现它的图处理框架 Pregel 时借鉴了 BSP 的一些理念。而 Spark 自己的 Pregel API 也受到了谷歌 Pregel 的启发。

如图 4.12 所示，算法被分成一系列超步（superstep），每一个超步就是一轮单独的迭代。在每个超步内部，每个顶点的计算都是并行的。在超步结束前，每一个顶点都为其他顶点生成消息，这些消息传递到下一个超步中。由于有同步屏障（synchronization barrier）机制，直到当前超步全部执行完后，下一个超步才会开始执行。与通用的分布式高性能计算底层库不同，GraphX 为 Pregel API 提供了自动同步机制。

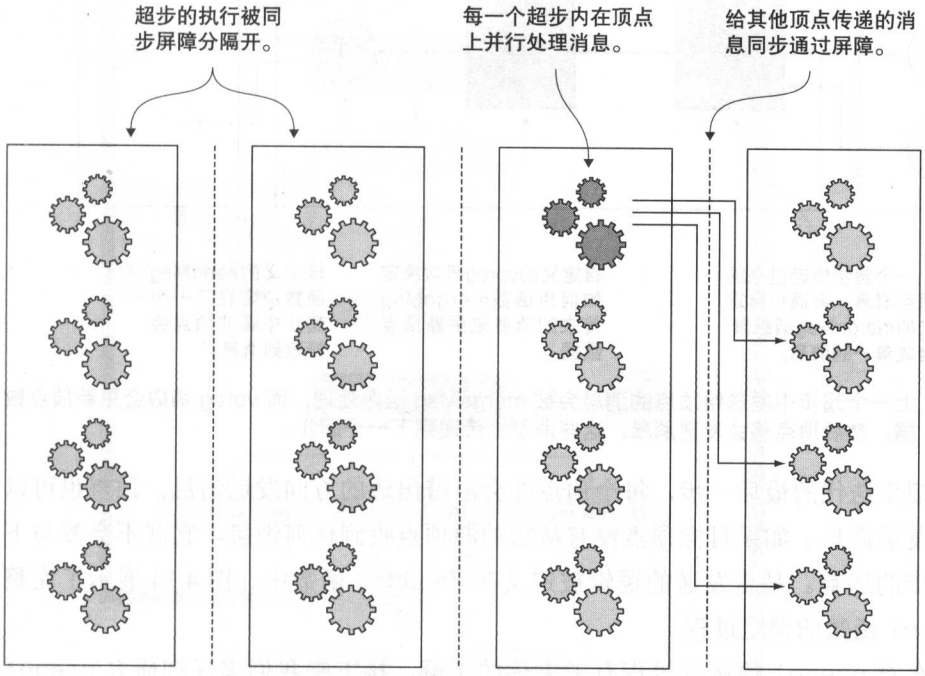


图 4.12 BSP 允许在一个超步内并行化处理消息，然后生成消息并传递给下一个超步。直到当前超步结束并将所有消息传给下一个超步后，下一个超步才会开始。这就是同步屏障。

上述框架的一个好处是，它具有高层次的抽象，允许开发者指定相关的程序行为，并由框架进行高效的并行化执行。

图 4.13 展示了 Pregel API 完成一个超步的内部细节。上一个超步发送过来的消息会被顶点聚集在一起，然后由“消息合并”（mergeMsg）函数进行处理，这样每个顶点就只处理一条合并后的消息（除非没有消息发送给这个顶点）。mergeMsg 函数运行的方式与 aggregateMessages 的 mergeMsg 函数完全一致：它必须是可交

换的、可组合的，这样就可以通过反复处理消息对来为每个顶点生成单一的结果数据。

与 aggregateMessages 不同，mergeMsg 函数返回结果消息但不会直接对顶点进行更新，它会把返回的结果消息作为参数传递给顶点处理程序 vprog，vprog 以顶点（包括顶点的 VertexID 及其数据）和消息作为输入，返回新的顶点数据以便被框架更新到顶点中。

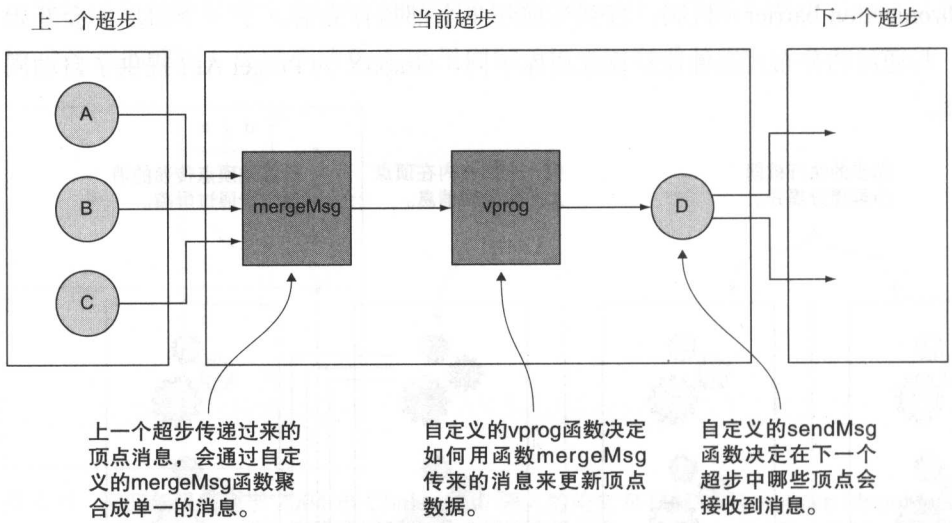


图 4.13 上一个超步中发送给顶点的消息会被 mergeMsg 函数处理，而 vprog 函数会更新顶点数据，然后顶点将会发送消息，这些消息会传递到下一个超步。

在超步迭代的最后一步，每个顶点都会沿着出边的方向发送消息。顶点也可以选择不发送消息；如果目标顶点没有从它的源顶点收到任何消息，它就不会参与下一个超步的运算。是否发送的逻辑被定义在 sendMsg 函数中。图 4.14 展示了完整的 pregel 函数的调用过程。

现在对 pregel 的运行过程有了大体的了解，接下来我们来仔细研究 pregel 函数的意义：

```
def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED]
```

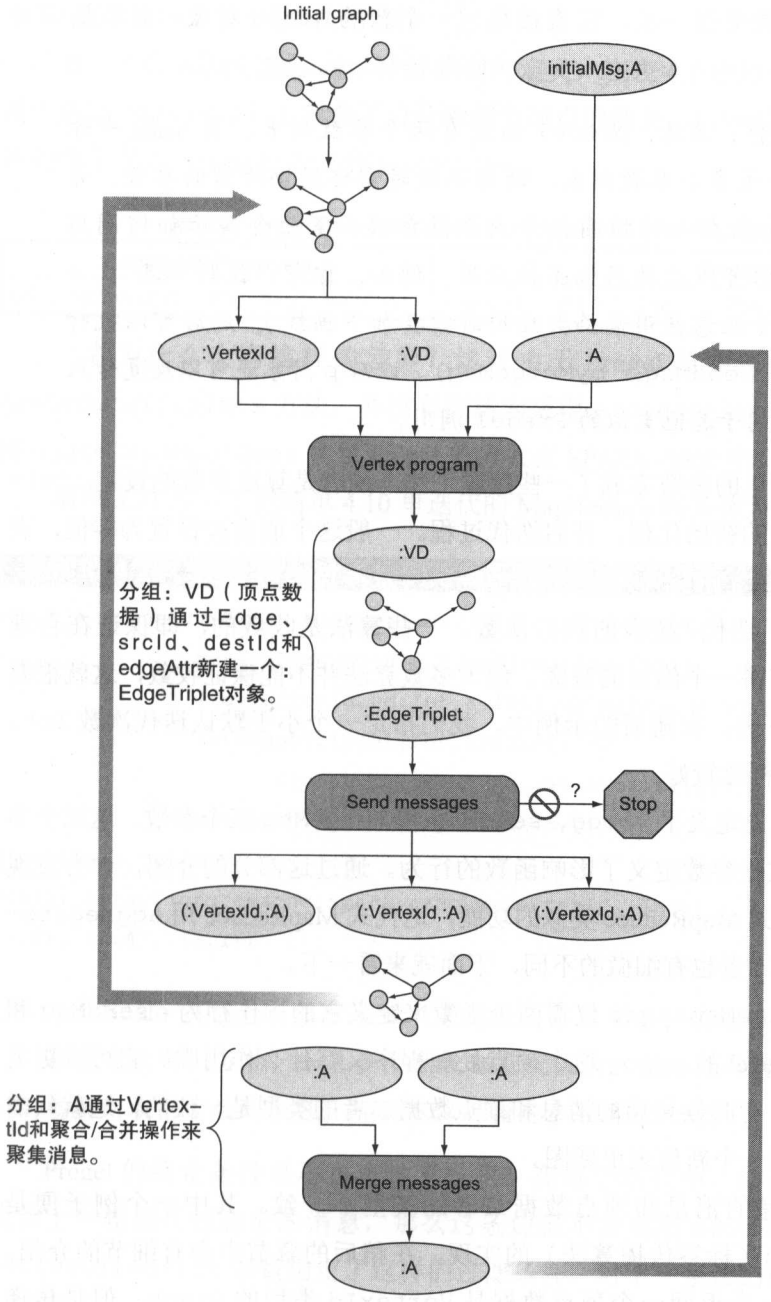


图 4.14 GraphX Pregl API 的执行流程。

如同 aggregateMessages 一样，pregel 是一个由消息（A）决定的参数化类型函数，A 类型的消息会被每个顶点接收和传递。而与 aggregateMessages 不

同的是，pregel函数更进一步，它直接返回一个新的Graph对象，而不是一个VertexRDD对象。

> **Scala小贴士：**注意，pregel函数有两个参数列表。在Scala中你可以将参数拆分成多个参数列表。调用函数时需要提供所需的参数，并且要按照语法要求在不同的圆括号内设置参数。下面会演示如何调用pregel函数。你可以选择用偏函数应用。例如，你可以执行val p = Pregel(g,0)_（注意在用偏函数应用时需要加下画线），然后可以执行p(myVprog, mySendMsg, myMergeMsg)。以后p对象还可以反复传入第二个参数列表用于其他类似的Pregel调用。

第一个参数列表里的参数完成了一些配置工作，或者是算法参数的设定。initialMsg是传给顶点的初始化值，开启迭代过程。一般这个值会被设置为零值，表示在算法开始时缺少相关的信息。

maxIter定义了迭代/超步的执行次数。一些算法是收敛的，即保证在合理的迭代次数后可以获得一个确定的答案。但大多数算法并不能保证收敛，这就很有可能算法永远迭代下去。在随后的示例中，我们指定一个小于默认迭代次数Int.MaxValue的值，这样比较好。

第二个参数列表里定义了vprog、sendMsg和mergeMsg三个参数，这三个参数也是三个函数。这些参数定义了影响函数的行为。通过这部分的介绍，你会发现Pregel提供了和迭代式MapReduce类似的功能，迭代式MapReduce用aggregateMessages实现。但二者也有细微的不同，下面就来看一下。

首先，aggregateMessages仅需两个函数来定义它的操作行为：sendMsg和mergeMsg。那么Pregel的vprog这个顶点处理程序又有什么作用呢？它可以更灵活地定义处理逻辑。有时候传递的消息和顶点数据二者的类型是一致的，这就只需要一些简单的逻辑用一个新值来更新图。

而有时候，传递的消息和顶点数据二者的类型不一致。其中一个例子便是LabelPropagation（标签传播算法）的实现，在稍后的章节中会有细节的介绍。LabelPropagation会返回一个顶点数据是VertexId类型的Graph，但是传递的消息类型则是VertexId到Long的Map类型——Map[VertexId, Long]。传递给顶点程序vprg的Map消息会用于找到有最大Long值的VertexId，然后这

个 ID 就会被用于顶点数据的更新。

另一个明显的区别是 sendMsg 函数的定义，在表 4.2 中对它们进行了比较。前面提到过 EdgeTriplet 包含了边及其两个顶点的消息。EdgeContext 还包含了额外的两个方法，sendToSrc 和 sendToDst。

表 4.2 aggregateMessages 和 Pregel 中的 sendMsg 函数对比

aggregateMessages	Pregel
EdgeContext[VD, ED, Msg] => Unit	EdgeTriplet[VD, ED] => Iterator[(VertexId, A)]

二者为什么会有这样的差异？这是由于 Pregel 的实现仍然依赖已废弃的 mapReduceTriplets 方法，并没有更新为采用 aggregateMessages 的版本。采用 aggregateMessages 实现 Pregel 的方式在 SPARK-5062 中有介绍。

清单 4.21 展示了把清单 4.10 里迭代的 MapReduce 例子转为用 Pregel 实现。

清单 4.21 使用 Pregel 来找到距离最远的顶点

```
val g = Pregel(myGraph.mapVertices((vid,vd) => 0), 0,
               activeDirection = EdgeDirection.Out)(
               (id:VertexId,vd:Int,a:Int) => math.max(vd,a),
               (et:EdgeTriplet[Int,String]) =>
                     Iterator((et.dstId, et.srcAttr+1)),
               (a:Int,b:Int) => math.max(a,b))
g.vertices.collect
res12: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((4,3), (1,0),
(3,2), (5,4), (2,1))
```

Scala 小贴士：你永远可以考虑使用具名参数，有时候这会让代码更具有可读性。另外，如果多个参数都有默认值，在函数调用时要是只用到参数列表中靠后的参数，那就可以对指定参数名传入值。

Pregel 的终止条件是不再有消息发送。在每个迭代中，如果边上的顶点没有收到上一轮迭代传递来的消息，那么这条边就不会调用 sendMsg 函数。Pregel 的 activeDirection 参数指定了这样的过滤条件。对于一条边的两个顶点 srcId 和 dstId：

- EdgeDirection.Out——当 srcId 收到来自上一轮迭代的消息时，就会调用 sendMsg，这意味着把这条边当作 srcId 的“出边”。

- EdgeDirection.In——当 dstId 收到来自上一轮迭代的消息时，就会调用 sendMsg，这意味着把这条边当作 dstId 的“入边”。
- EdgeDirection.Either——只要 srcId 或 dstId 收到来自上一轮迭代的消息时，就会调用 sendMsg。
- EdgeDirection.Both——只有 srcId 和 dstId 都收到来自上一轮迭代的消息时，才会调用 sendMsg。

在顶点距离的例子中，我们采用了 EdgeDirection.Out，因为顶点距离算法是按照图中边的方向进行计算的。一旦它到达了有向图的“终点”，算法终止。

如果图有环，就需要小心地处理，因为这时有可能会有一部分顶点不断发送和接收消息，导致算法永远不会终止。解决这个问题的一个方法是用 maxIterations 参数来保证算法会在一个合理的迭代次数后终止。

另一个方法是尝试去检测算法是否陷入了不断循环却没有实际更新的状态。例如，下一章我们会介绍最短路径算法，它会不断更新从当前顶点可到达的其他顶点的映射表。只有在映射表有新消息增加的情况下，sendMsg 才会执行发送消息的动作。

4.6 小结

- Pregel 和 aggregateMessages() 函数是 GraphX 图处理的基础，二者有细微的差别。
- 大部分 GraphX 的内置算法是用 Pregel 实现的。
- 由于 Pregel 的终止条件是不再有需要发送的消息，所以要求有更灵活的终止条件的算法可以用 aggregateMessages() 实现。
- 由于 GraphX 没有内置的用于读写属性图的 API 函数（只有读取边列表的函数），我们用示例代码演示了如何读写属性图。
- 如果没有合适的数据，GraphX API 可提供生成随机图和确定图的方法。

5 内置图算法

本章要点

- GraphX API 中的内置图算法
- 集群中的图探测算法：PageRank，最短路径，连通组件，标签传播
- 用三角形衡量图或子图的连通性
- 衡量社交网络图中的用户子集的连通性，找出孤岛人群

在第 4 章，我们已经了解了 GraphX API 的基础知识，现在可以写一些自定义的图算法了。但是，如果 GraphX API 已经提供了标准算法的实现，那么就没有必要重新造轮子了。本章主要介绍这些基础算法及其适用场景：

- 网页排名（PageRank）
- 个性化的网页排名
- 三角形数
- 最短路径
- 连通组件

- 强连通组件
- 标签传播

第7章会介绍一个更加有用的高级内置图算法：SVDPlusPlus。

5.1 找出重要的图节点：网页排名

第2章介绍了网页排名PageRank算法，它最初是搜索引擎用于页面排名，也可以用于在论文引用关系网中查找最有影响力的论文。通常PageRank是在图中查找最“重要”的节点。接下来我们深入研究PageRank算法：不同的场景、不同的参数以及不同的调用方式，PageRank会如何做。注意，PageRank专利属于斯坦福大学，商标属于Google。

5.1.1 PageRank算法解释

PageRank是衡量图中顶点的权值的一种方法。在第2章的一个示例中，我们根据书目引文的集合度量了科学论文的影响力。虽然PageRank的最初应用场景是给网页爬虫引擎爬取的页面分配一个权值，但是PageRank算法确实可以用于任何有向图中每个顶点权值的计算。其他PageRank的应用包括，在基于人到人的关系图中通过权值排名区分出关键人群，在基于“分享”的社交网络图中利用一些高级机器学习技术对影响力做等级划分，如协同过滤和语义关联。

PageRank的一个最简单用法是计算每个顶点的入度，类似于4.2.2节我们计算出度的方式。GraphX API提供的`outdegrees()`函数计算出度不需要任何额外的代码。许多人错误地认为，这样的计算“入链”是PageRank的全部，其实PageRank能做更多的事情。

PageRank寻求优化的递推公式，并不是基于问题域中边所指向顶点的数目，而是基于这些顶点的页面排名。

虽然PageRank算法的定义是一个递归调用，但PageRank的算法实现却是直接的迭代计算。如图5.1所示，改编自1999年佩奇和布林的PageRank论文《网页引用排名：给网页排序》，插图解释了该算法。

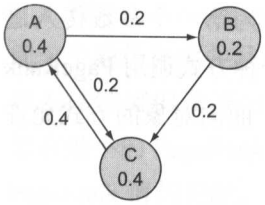


图 5.1 PageRank 迭代。这个特定的退出迭代的条件是：当前迭代是算法计算结果最稳定的且是最终状态，因为 PageRank 在顶点上重新分布计算，顶点排名的权值不再变化，即稳定状态。

算法描述如下。

1 用 1/*N* 的页面排名值初始化每个顶点，*N* 是图中顶点总数。

2 循环：

- 每个顶点，沿着出边发送 PR 值 1/*M*，*M* 为当前顶点的出度。
- 当每个顶点从相邻顶点收到其发送的 PR 值后，合计这些 PR 值后作为当前顶点的新 PR。
- 图中顶点的 PR 与上一个迭代相比没有显著的变化，则退出迭代。

5.1.2 在 GraphX 中使用 PageRank

正如在第 2 章所见，GraphX 已经实现了 PageRank 算法，我们没必要再敲一遍先前小节中介绍的算法代码。在这一节中，你会看到调用 PageRank 的两种不同方式以及算法的参数设置。

面向对象方式调用 PageRank 与基于对象方式调用 PageRank

GraphX 提供了两种调用 PageRank 的方式：面向对象和基于对象。在第 2 章中，我们在 Graph 对象上调用了 pageRank() 函数，这是面向对象的方式。如在 4.2.2 节中看到的，pageRank() 是 GraphOps 的一个函数，Graph 对象自动创建一个 GraphOps 对象并提供一个在需要的时候自身转化成 GraphOps 的方法，在需要的时候使用同样的方式，一个 RDD 也会转化成一个 PairRDD。图 5.2 显示了不同类之间的关系。

提示：要找出 Graph 实例所有可用的函数，除了查 Graph 文档还要查 GraphOps API 文档。

PageRank 的另一种调用方式是基于对象的，即调用单例对象 org.apache.

spark.graphx.lib.PageRank 的 run() 方法，把图对象作为第一个参数传入进来，这和 GraphOps 的 pageRank() 方法的做法一样。用哪一种方式调用 PageRank 算法只是一个风格问题，例如，已经在图上执行了许多操作，面向对象的方式允许你把它作为一个额外的操作链。

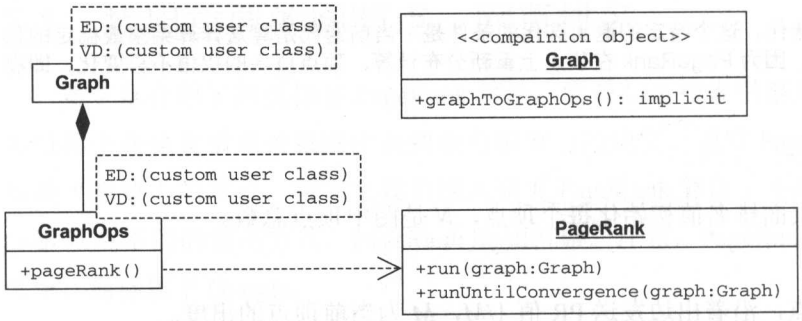


图 5.2 伴生对象 Graph 包含隐式转换方法，从 Graph 对象转换成相应的 GraphOps 对象，所以所有 GraphOps 中的操作都可以像在 Graph 对象中一样被调用。GraphOps 包含 pageRank() 方法，它从单例对象 PageRank 中调用 call() 方法，作为第一个参数传递到图中。

固定的迭代次数（静态）与公差的退出条件（动态）

对于这两种调用方式（面向对象和基于对象），还要做个选择：是在指定的迭代次数后退出，还是永远迭代下去直到满足一个退出条件后再退出。GraphX API 文档中称第一种为静态退出，后一种为动态退出。

静态退出方式传入一个 numIter 参数（迭代的次数），动态退出方式传入 tol 参数（公差）。如果在上一次迭代和当前迭代间，顶点的 PageRank 值的变化小于 tol 公差，应用将会从算法中跳出，不发送顶点的 PageRank 值到相邻顶点也不关注相邻顶点发送过来的 PageRank 值。tol 用于决定算法什么时候终止：如果全图中的顶点变化值全部小于 tol，算法终止。

在第 2 章的 PageRank 实例中，tol 的值是 0.001，这是一个比较大的值，适用于快速终止算法的场景。如果要更精确的结果，tol 就要取较小的值，如 0.0001。

随机重置概率

这 4 种变化（面向对象与基于对象，静态与动态）都需要一个参数 resetProb，在 API 文档中也被称为 alpha。这个 resetProb 参数与 1998 年佩奇和布林的论文（《大规模超文本 Web 搜索引擎的分解》）相符，又称为抑制因子。指定 resetProb 在 [0,1] 范围内，表示最小 PageRank 值的分类方式。

理论上，resetProb 符合这样一个几率：一个 Web 用户在一个页面上突然访问一个随机页面并且该随机链接并不包含在用户当前所在的页面上，这对于计算具有入站链接但没有出站链接的 sink-web 页面很有用。resetProb 确保所有的页面都有最小的 PageRank，同样前面的设定值（1-resetProb）抑制从相邻顶点传入的 PageRank 值的贡献。这就像添加了虚构的从 sink 顶点指向图中其他边的连出边，这是为了保证公平，同样的也会适用于无出站的顶点。

> **注意**：GraphX 用到的 resetProb 与 PageRank 论文中的 1-*d* 意思相同，*d* 表示抑制因子。PageRank 论文中推荐抑制因子为 0.85，GraphX 文档中推荐 resetProb 为 0.15。

下面的公式包含了 resetProb 计算顶点 v 的新 PageRank 值：

$$v'_{PageRank} = resetProb + (1 - resetProb) \sum_{\text{u到v的出边}} \frac{u_{PageRank}}{u_{Outdegree}}$$

resetProb 从某种意义上来说就是一个修正。对于一个真正的理想的 PageRank，resetProb 应当被设置为 0，然而这样会导致过长的收敛时间也会导致一个退化的结果：自连接的集群部分取得全部的 PageRank，主流高度连接顶点得到的 PageRank 为 0。示例如图 5.3 所示。

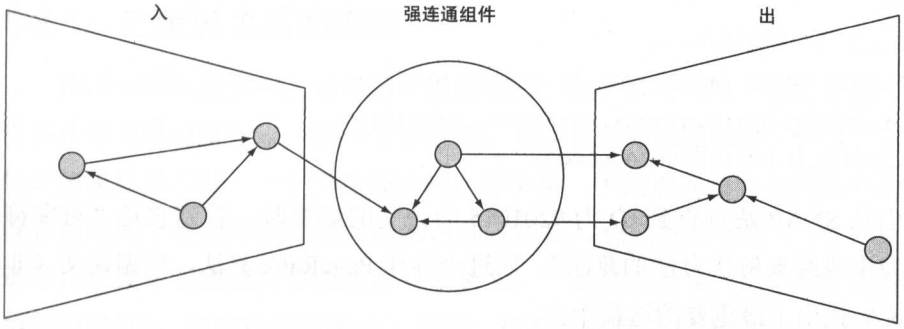


图 5.3 真实世界中的 Web 页面的图据说类似于蝴蝶结。用 1.0 的抑制因子（resetProb 为 0.0），所有的页面排名落在“出”顶点集合中，所有的“入”顶点集会由于 PageRank 为 0 而被丢弃。

基于 resetProb 的含义，你会觉得需要一个随机数生成器。虽然一些 PageRank 算法的实现采用了随机数生成器，但在 GraphX 的实现中 resetProb 使用严格的确定的常数。采用随机数生成器的 PageRank 算法实现的一个例子是，用蒙特卡罗法（Monte Carlo technique）耗费较少的计算时间来估计 PageRank。

5.1.3 个性化的 PageRank

假设不是对 Web 页面进行排名，而是在一个社交网络中给用户推荐新人。这样的推荐是对正在查找其他用户的定制，这个用户更可能对在图中离他不太远的用户感兴趣。

个性化 PageRank 是 PageRank 的一个变种，给图中指定的“源”顶点一个相对权值。从概念上讲，如上一部分所说的虚构的 Web 用户（社交网络图中的用户），当这些用户突然决定访问另一顶点，会一直落在指定的源顶点。在 GraphX 中，这个虚构的用户场景是这样实现的，通过在指定的源顶点上强制一个最小的 PageRank 值，其他顶点的 PageRank 值允许跌为 0（假如顶点没有入链）。

在第 2 章我们看到 PageRank 应用在论文引用网络中，在那个示例中，论文《非对称弦理论》拥有最大的 PageRank 值。在下面的清单 5.1 中，我们指定源顶点是 9207016，即论文的 ID。

清单 5.1 个性化 PageRank 找出最重要的关联论文

```
import org.apache.spark.graphx._
val g = GraphLoader.edgeListFile(sc, "cit-HepTh.txt")
g.personalizedPageRank(9207016, 0.001)
 .vertices
 .filter(_._1 != 9207016)
 .reduce((a,b) => if (a._2 > b._2) a else b)
res1: (org.apache.spark.graphx.VertexId, Double) =
       (9201015,0.09211875000000003)
```

当我们从 SNAP 站点查到 ID 为 9201015 的论文的摘要时，看到它是“对字符串高效能力生成经典解决办法的算法”。通过个性化 PageRank 算法，根据论文《非对称弦理论》找出了最重要的这篇论文。

个性化 PageRank 的 GraphX 实现与其他实现相比在多个方面都是受限制的。首先，只有一个源顶点可以被指定。如果允许指定一组源顶点，这将可以找到对一组人来讲最重要的人，如 1992 年的哈佛校友。其次，不指定每个源顶点的权重值，这在 GraphX 的实现中硬编码为 1，意味着一个顶点的最小值是两个极端之一：0 用于非源顶点，或源顶点为 `1*resetProb`。当 GraphX 只允许指定一个源顶点，这不算是大的局限，而当 GraphX 未来要指定多个源点并要为每个源顶点指定互相独立的

权重值时，这就要允许用户在源顶点集合中对其余的源顶点指定某种密切度或重要度。

5.2 衡量连通性：三角形数

当 PageRank 度量单个顶点的影响力时，通过计算三角形数可以衡量图或子图的连通性，也就是顶点如何共同互相影响。总的来说，顶点在一起相互影响。例如，在一个社交网络中，如果每个人都影响其他人——如果每个人都连接到其他人——这会产生大量的三角形关系。

听起来一个三角形关系像是：三个顶点，它们的边是全连接的。但是 GraphX 在处理有向图时有些细微的差别。在对三角形计数时，GraphX 把图当作无向图，忽略边的方向（参见图 5.4），重复的边合并为一个，忽略方向同时也会消除从一个顶点指向自身的循环边。

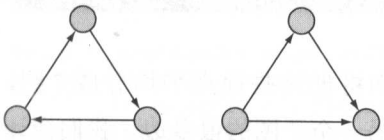


图 5.4 在计算三角形关系时，GraphX 并不关心边的方向。这两个都是三角形关系，即使左边的三角形构成了一个环，右边的三角形有一个死胡同。

5.2.1 三角形关系的用法

图或子图有越多的三角形则连通性越好。这个性质可用于确定小圈子（图中有很多互相关联的部分），可用于提供推荐，也可以识别垃圾邮件。但是，有时候连通性好效果却并不好。一个全连通的图，其中每一个顶点连接到其他所有的顶点，却没有传递关于连通性的信息（虽然边和顶点属性仍然携带信息）。有时缺乏连通性却可以在图中识别出有价值的顶点，例如，一篇研究论文（《网络讨论组中的社会角色的特征》，出自韦尔泽等人）表明，那些在在线论坛上经常回答问题的人通常是一些独来独往的人，与其他的论坛参与者有较弱的联系，这就导致这些人在图中的关系很少有三角形。这些乐于助人的问题解答者可能与提问者有许多垂直的边连接关系，这些问答者之间并不是彼此互相连接的。所以这些独来独往的问题回答者不属于任何密集网络人群，所以不会形成几个三角形关系（只有当提问者之间彼此相连的时候才会有许多三角形关系形成）。当试图确定有价值的问题的解答者，一个标志或许是具有较少的三角形关系。

三角形数可作为聚类系数和传递率的一个因子。这些都是更复杂的计算，因为它们不仅仅涉及计算三角形关系数。通过用分母来标准化，可以很容易比较大小不同的图的连通性。在 Spark 1.6 版本中，GraphX 没有内置算法来计算聚类系数和传递率。第 8 章展示了如何计算全局聚类系数。

5.2.2 Slashdot 朋友和反对者的用户关系示例

在这里，我们将展示使用三角形度量任意 Slashdot.org 用户子集之间的连通性，Slashdot.org 是现在流行的科技新闻和讨论的网站，创建于 1997 年。然而，我们在第 2 章用的同样的斯坦福图数据，也有一个匿名的 Slashdot 的“朋友”和“反对者”边列表。在 Slashdot 上，用户阅读评论时可以标记评论的作者是朋友还是反对者，这样就可以在下次遇到这个评论者写的评论时有个提醒。即使 Slashdot 的数据是匿名的，即顶点 ID 不会匹配到现实的 Slashdot 用户 ID，匿名顶点 ID 似乎仍然在持续增加；而长期的用户会减少顶点 ID 总量。

在这些图数据中，顶点 ID 从 0 到 70,000。我们将把这些顶点平均分成 7 组，每组 10,000 个顶点。这意味着会失去从一个子图到另一个子图的很多边。我们会计算每个子图的三角形数量，看看是否有随着时间推移的趋势变化。由于老用户往往倾向于和其他老用户交流，这应该有很高的关联度。我们希望看到有 10,000 个顶点的第一个子图是一个紧密的高度连接的群组，第二个子图的连接分为内部顶点之间的连接以及连向第一个子图中受人尊敬的用户的连接。但是要注意，我们切分子图时会丢弃那些从第二个子图到第一个子图的临界关系的边。以此类推，我们期望第三个子图与其他子图间有更少的三角形关系。

GraphX 的子图

GraphX 的子图用起来很简单。`subgraph()` 函数有两个参数：边判定函数和顶点判定函数。两者都不是必需的，你可以指定一个。边的判定函数推导图的每条边，函数返回 `true` 或 `false`。`true` 表示这条边是子图的一部分。顶点判定函数也具有同样的功能。边判定函数过滤掉子图中所有孤立顶点所在的边，顶点判定函数过滤掉子图中孤立边所包含的顶点。

注意：在 Spark 1.6 及更早的版本中，`triangleCount()` 函数强加了几个严格的前提条件：第一，图要选择一个在 9.4 节中描述的

PartitionStrategy 分区策略。第二，如果有任何重复的边（两个或两个以上的相同的两个顶点之间的边），那些重复的边必须指向同一个方向。GraphX 文档中夸大了第二个要求条件，文档中说所有边必须规范有序，即从小的顶点 ID 指向大的顶点 ID。通常最简单的办法是改造图满足第二个要求，但如果你的图没有重复的边，那就没有什么可担心的（只需满足第一个分区的要求即可）。Jira SPARK-3650 尚未明确这些要求在哪个 Spark 发行版本中解决（截至 Spark 1.6）。

接下来我们开始验证这个想法，首先从 http://snap.stanford.edu/data/soc-Slashdot0811.html 下载 Slashdot 的关系数据，然后解压缩。进入 Spark Shell，输入清单 5.2 中的代码。

清单 5.2 Slashdot 用户的三角形关系

```
val g = GraphLoader.edgeListFile(sc, "soc-Slashdot0811.txt").cache
val g2 = Graph(g.vertices, g.edges.map(e =>
       if (e.srcId < e.dstId) e else new Edge(e.dstId, e.srcId, e.attr))).
    partitionBy(PartitionStrategy.RandomVertexCut)
(0 to 6).map(i => g2.subgraph(vpred =
       (vid,_) => vid >= i*10000 && vid < (i+1)*10000).
    triangleCount.vertices.map(_._2).reduce(_ + _))
res1: scala.collection.immutable.IndexedSeq[Int] = Vector(1352001, 61376,
10865, 3935, 1384, 786, 658)
```

构建一个scala.collection.immutable.Range对象的简写方式。Range调用map()函数是函数式循环的用法。

vpred是subgraph()函数的第二个参数，这里略过第一个参数（用第一个参数的默认值），所以就需要指定第二个参数的参数名。

```
(0 to 6).map(i => g2.subgraph(vpred =
       (vid,_) => vid >= i*10000 && vid < (i+1)*10000)
```

这里的下画线是Scala的通配符用法，表示这个参数在这里不需要，可以忽略。

匿名函数有两个参数，看起来像Tuple2，但这的确是Scala带两个确定参数的匿名函数的写法。

```
triangleCount.vertices.map(_._2).reduce(_ + _))
```

triangleCount()函数实际上是基于每个顶点计算三角形关系，返回一个Graph[Int,ED]对象。然后合计所有顶点的三角形关系数（保存在顶点属性中，通过 _._2获取）。

Scala 小贴士：Scala 允许两种不同的函数调用语法。一种是 Java 风格，句号在前，参数在圆括号中。另外一种是省略了句号和括号的方式。如果函数没有参数（称为后缀表达式）或者有一个参数（称为中缀表达式），Scala 允许省略句号和括号。风格上而言，Scala 程序员一般会尽可能地省略括号，特别是在函数没有副作用的时候（函数中引用到的对象是不变的）。在前面的代码图解中，to 是 Scala.Int 中的一个函数（to 前面的 0 是一个对象实例），而后面的 6 是 to 的参数。to 不是 Scala 的保留字而只不过是 Scala 标准类库中的一个函数。to 函数的 API 文档也可以在 Scala.Int 中找到。

g2 的计算部分是为了确保顶点 ID 是升序排列的，如果 SPARK-3650 被合并进去后，这块的代码也就没必要了。

我们的假设被证实了：每个含有 10,000 个顶点的子图具有较低的三角形数。

在这个简单的示例中，我们并不关心每个顶点上的三角形关系数，但是要考虑不同于图全局连通性的本地连通性，这类信息很有用。

与 PageRank 一样，在 graphx.lib 中也有一个基于对象的 TriangleCount 版本。

5.3 查找最少的跳跃：最短路径

GraphX 中内置的最短路径算法用来计算跳跃数（不用边上的任何距离属性值），并依据跳跃顺序返回距离（不是如何从一个顶点到另一顶点的全路径）。

看到这个名字，你可能会误解为要在映射中绘制一个路由图，这样的算法在 6.2 节会看到。

这里举个最短路径算法的例子，要计算跳跃数就是要计算从社交网络图中的每个顶点到达顶点“弗雷德·马普尔”的最小的“朋友”边数。换句话说，由于 GraphX 支持传入一组顶点，如熟知的 landmarks，那么从图的每个顶点到这组顶点的最短距离就可以计算了。例如，任何在 79 级的同学。另一个例子是，在计算机网络中计算到最近的一级网络节点的最小跳跃数。

下面举一个简单的例子，用图 1.5 中的示例，在 Spark Shell 中运行清单 4.1 中的代码，代码见清单 5.3，计算从图中每个顶点到 Charles 顶点的最小跳跃数。注意，虽然算法正式命名为“最短路径”，但从 API 中可以看出，GraphX 仅仅是返回最短

的距离。计算结果见图 5.5。

清单 5.3 计算最短路径

```
lib.ShortestPaths.run(myGraph,Array(3)).vertices.collect
res2: Array[(org.apache.spark.graphx.VertexId,
    org.apache.spark.graphx.lib.ShortestPaths.SPMap)] = Array((4,Map()),
    (1,Map(3 -> 2)), (3,Map(3 -> 0)), (5,Map()), (2,Map(3 -> 1)))
```

注意，ShortestPaths 只能用基于对象的方式调用，因为在 Graph 或 GraphOps 中没有相应的方法。

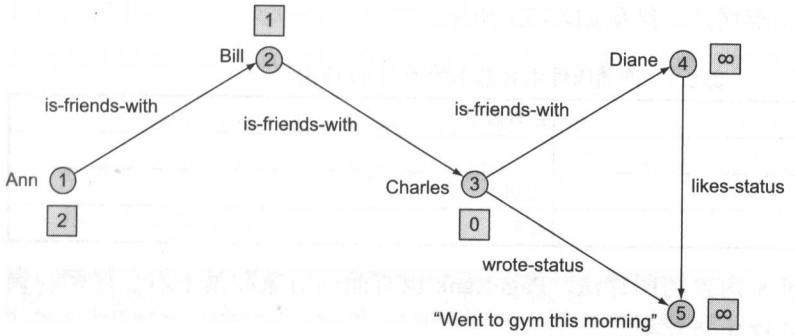


图 5.5 ShortestPaths 要找到从图中每个顶点到指定顶点（这里是顶点 #3）的跳跃数（图中方框内的数字）。这个算法不会使用任何边权重信息（表示图中的距离值），它仅仅返回从一个顶点到目标顶点的跳跃数，没有任何关于如何达到最短距离的路由信息。

5.4 找到孤岛人群：连通组件

连通组件能在社交网络图中找到一些孤立的小圈子，并把它们在数据中心网络中区分开。连通组件算法与有向图和无向图都有关联。构造一个如图 5.6 所示的图，在下面的清单 5.4 中找出其中的连通组件。

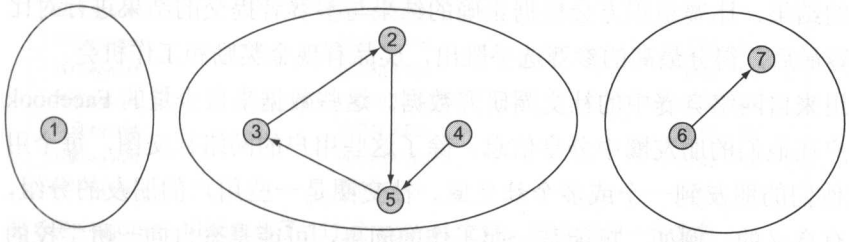


图 5.6 图由 7 个顶点和 5 条边构成，其中有三个连通组件。假如这些图数据来自于社交网络，那么每个连通组件都可以认为是一个小圈子。

清单 5.4 执行 connectedComponents()

```
val g = Graph(sc.makeRDD((1L to 7L).map((_,""))),
    sc.makeRDD(Array(Edge(2L,5L,""), Edge(5L,3L,""), Edge(3L,2L,""),
                     Edge(4L,5L,""), Edge(6L,7L,""))).cache
g.connectedComponents.vertices.map(_.swap).groupByKey.map(_._2).collect
res3: Array[Iterable[org.apache.spark.graphx.VertexId]] = Array(
CompactBuffer(1), CompactBuffer(6, 7), CompactBuffer(4, 3, 5, 2))
```

connectedComponents 函数返回一个与输入的图对象结构相同的新 Graph 对象。连通组件是用其中最小的顶点 ID 标示的，而这个最小的顶点 ID 会赋值给这个连通组件中的每个顶点属性。结果如表 5.1 所示。

表 5.1 连通组件由其最小的顶点 ID 标识

组件 ID	组件成员
1	1
2	2,3,4,5
6	6, 7

正如其他 GraphX 内置的图算法，PageRank 也有面向对象和基于对象这两种调用 GraphX 连通组件算法的方式。

5.4.1 预测社交圈子

刚才我们看到在 GraphX 中生成一个连通组件是多么简单，现在我们把算法应用到真实的数据集上。我们不仅要运行刚才看到的连通组件算法，还要看看如何导入数据，如何将它们转成我们所需的数据结构，输出特定格式的结果数据。用到的数据集来自于 Kaggle 2014 数据科学竞赛中 Facebook 的数据集。

在 Kaggle（www.kaggle.com）举办的竞赛上，会提供数据集下载，而参赛者需要设计一个任务来为数据集中的每条记录预测一个确定的结果。在竞赛过程中，参赛者提交预测的结果，比赛组织方会根据正确的结果与参赛者提交的结果进行对比并打分。在竞赛最后，得分最高的参赛选手胜出，奖品有现金奖励和工作机会。

我们将使用来自网络竞赛中的社交圈研究数据。这些数据来自少量的 Facebook 用户，这些用户在他们的朋友圈中分享信息。除了这些用户的网络社交图，每个用户被要求分配他们的朋友到一个或多个社交圈。社交圈是一些用户的朋友的分组，对这些用户是有意义的。例如，可能是一起工作的同事，可能是来自同一所学校的校友，也可能是他们来往的一组朋友。由什么构成了一个社交圈是用户决定的。圈

子可以重叠，可以完全由一个或多个其他圈子所包含，甚至可以是空的。

竞赛的目的是使用网络信息来获得用户如何划分社交圈子的最佳预测。显然有多种办法解决这个问题，但我们将用一个简单的近似方法在每个用户有连接关系的图中找出连通组件。我们预测每个用户有什么圈子，以及如何把朋友分配到这些圈子中，然后得到最终预测的连通组件。

获取 Kaggle 提供的社交数据

要从 Kaggle 上下载社交网络数据，首先要有一个 Kaggle 账号，然后进入 http://www.kaggle.com/c/learning-social-circles/data 下载页面，这个下载页面中列出了许多文件，我们需要下载 egonets.zip，即使竞赛结束了也要选择同意竞赛规则。解压文件，浏览文件目录，进入 socialcircles/data 下面的 egonets 目录，如图 5.7 所示。

> **注意**：egonet 这个术语来自于斯坦福的朱利安·麦考利和尤雷·莱斯科夫的论文，论文中他们称独立用户为 egos，称用户的连接为 alters，都很弗洛伊德学说化。

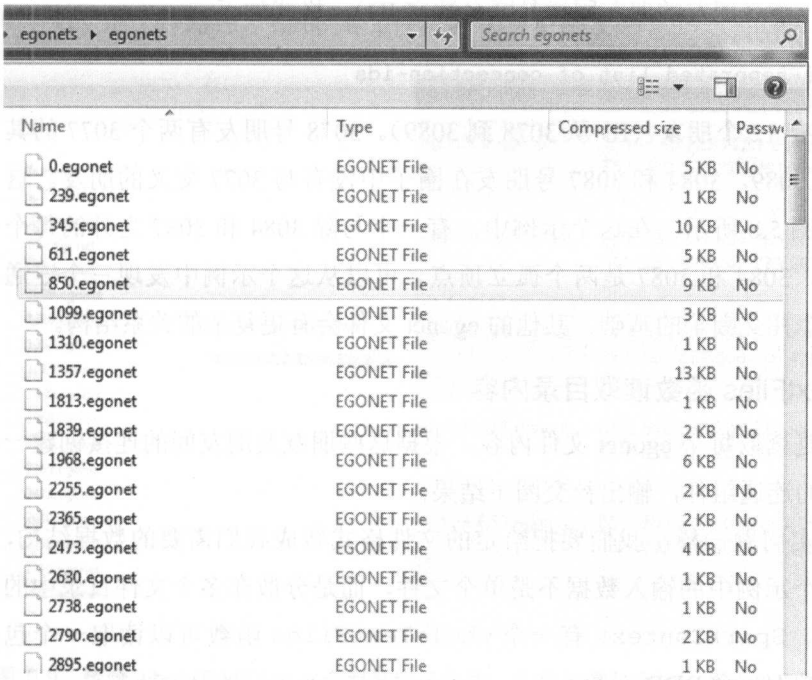


图 5.7 egonets 目录包含了 111 个 egonet 文件，每个用户一个文件。

数据集中的用户是匿名的，每个用户分配一个 ID，每个用户一个 egonet 文件。

打开一个文件，查看文件内容。选择内容较少能一页显示的3077.egonet文件，文件内容如下：

```
3078: 3085 3089
3079: 3082
3080: 3089
3081: 3085 3083 3089
3082: 3079 3086 3089
3083: 3085 3081 3089
3084:
3085: 3083 3078 3081 3088 3089
3086: 3082
3087:
3088: 3085 3089
3089: 3085 3080 3083 3078 3082 3081 3088
```

egonet文件列出了该用户的朋友，以及与他的朋友有连接的朋友列表，该用户的朋友一行展示一个（朋友的朋友同样是匿名数字ID），格式如下：

```
Friend-id: Space-seperated list of connection-ids
```

3077号用户有12个朋友（ID从3078到3089）。3078号朋友有两个3077的其他朋友，3085和3089。3084和3087号朋友在圈子中没有与3077交叉的朋友。这些图连接关系如图5.8所示。在这个示例中，有一个与除3084和3087之外的每个用户关联的大图，3084和3087是两个孤立顶点。可以从这个示例中发现三个连通组件，构成这三个社交圈子的基础。其他的egonet文件会有更复杂的关系结构。

用wholeTextFiles函数读取目录内容

我们的任务是读取每个egonet文件内容，根据这些朋友及朋友间的连接创建一个图，找出图中的连通组件，输出社交圈子结果。

就像其他现实问题一样，我们要把给定的文件格式转成我们需要的数据结构，创建一个图。这个示例中的输入数据不是单个文件，而是分散在多个文件目录中的文件。幸运的是，`SparkContext`有一个`wholeTextFiles`函数可以读取一个包含多个文件的目录为一个RDD对象：

```
val egonets = sc.wholeTextFiles("socialcircles/data/egonets")
```

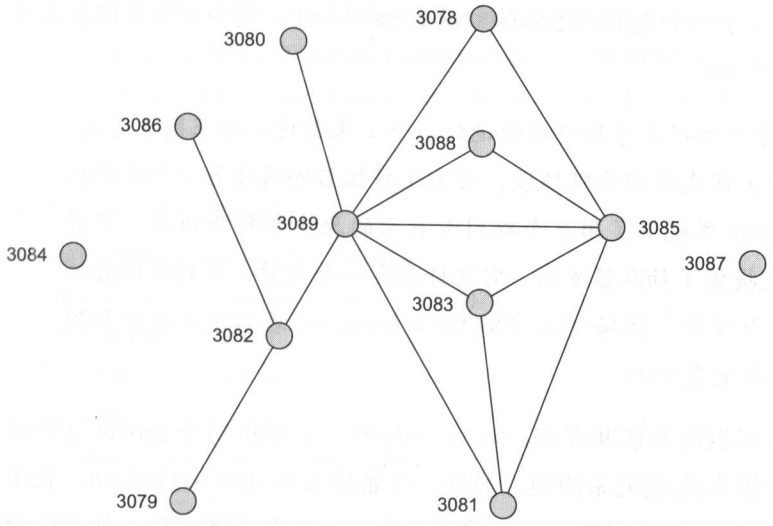


图 5.8 3077 号用户的 egonet 文件构成三个连通组件：两个孤立顶点组件（顶点 3084 和 3087），一个大的组件（所有其他顶点）。

`wholeTextFiles` 返回一个 `PairRDD` 对象，其中每个文件都是一个 key-value 时，key 是文件路径，value 是文件内容（参见图 5.9）

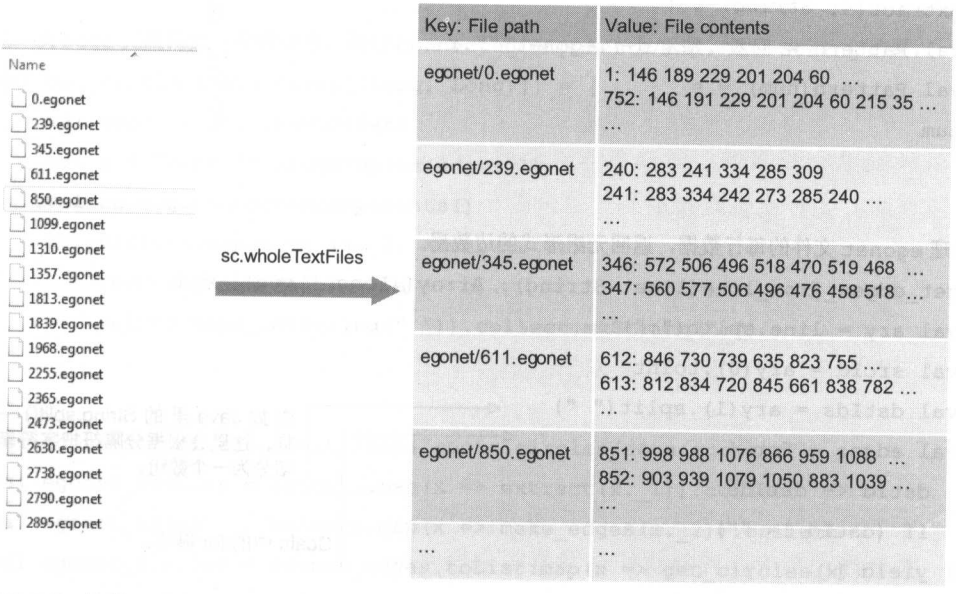


图 5.9 传给 wholeTextFiles 一个目录路径参数，它会生成一个 key 为文件路径、value 为文件内容的 PairRDD。这种方式读目录文件性能很好，同时也非常方便，最后所有的输入数据转成一个 RDD 对象。

下面我们首先定义了一个提取数据的映射函数 extract，其中采用正则表达式从文件名中提取出用户 ID。

> **Scala 小贴士**：可以在字符串后面追加一个 r 来创建一个正则表达式。在 Java 中写正则表达式通常比较烦琐，因为反斜杠需要通过另一个反斜杠来转义。而在 Scala 里提供了用三个双引号表示的原生字符串语法。同时 Scala 中的正则也提供了抽取器方法，它允许指定一个变量，正则匹配后的结果会赋值给这个变量。代码 val Pattern(num)=s 表示用正则匹配到字符串 s 后赋值给变量 num。

第二个映射表达式也称为辅助方法，make_edges，它解析每个 egonet 文件的内容以创建表示每个朋友连接关系的边。另外一个辅助方法 get_circles，它用 Graph.fromEdgeTuples 方法创建 egonet 的图对象，一旦有了图对象，就可以调用 connectedComponents 函数来获得社交圈子，代码如清单 5.5 所示。

清单 5.5 查找并且列出社交圈子

```
// 从 <path>/<userId>.egonet 格式的文件路径中解析出用户 ID。
def extract(s: String) = {
    val Pattern = """^.*?(\d+).egonet""".r
    val Pattern(num) = s
    num
}

// 处理 egonet 文件的每行数据，返回元组形式的边数据。
def get_edges_from_line(line: String): Array[(Long, Long)] = {
    val ary = line.split(":")
    val srcId = ary(0).toInt
    val dstIds = ary(1).split(" ")
    val edges = for {
      dstId <- dstIds
      if (dstId != "")
    } yield {
      (srcId.toLong, dstId.toLong)
    }
```

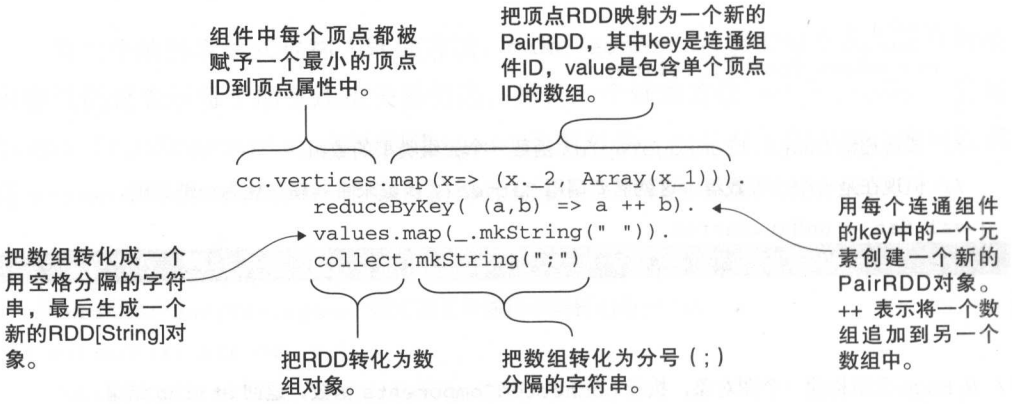
就如 Java 里的 String.split() 一样，这里会根据分隔符把字符串切分为一个数组。

Scala 中的 for 循环。

```
    // 一个细节：如果用户与其他任何人没有连接，则生成一个自连接，
    // 所以顶点会被包含在 Graph.fromEdgeTuples 构建的图里面。
    if (edges.size > 0) edges else Array((srcId, srcId))
}

// 从文件内容中构建边元组。
def make_edges(contents: String) = {
    val lines = contents.split("\n")
    val unflat = for {
      line <- lines
    } yield {
      get_edges_from_line(line)
    }
    // 要传递给 Graph.fromEdgeTuples 函数一个元组类型的数组。
    // 但现在是一个二维数组，这就需要调用 flatten() 函数来将其扁平化为一维数组。
    val flat = unflat.flatten
    flat
}

// 从 Edge 元组构建一个图对象，执行 connectedComponents 函数，返回 String 结果。
def get_circles(flat: Array[(Long, Long)]) = {
    val edges = sc.makeRDD(flat)
    val g = Graph.fromEdgeTuples(edges,1)
    val cc = g.connectedComponents()
    cc.vertices.map(x => (x._2, Array(x._1))).
        reduceByKey( (a,b) => a ++ b).
        values.map(_.mkString(" ")).collect.mkString(";")
}

val egonets = sc.wholeTextFiles("socialcircles/data/egonets")
val egonet_numbers = egonets.map(x => extract(x._1)).collect
val egonet_edges   = egonets.map(x => make_edges(x._2)).collect
val egonet_circles = egonet_edges.toList.map(x => get_circles(x))
println("UserId,Prediction")
val result = egonet_numbers.zip(egonet_circles).map(x => x._1 + "," + x._2)
println(result.mkString("\n"))
```

> **Scala 小贴士**：for 循环与 filter() 函数结合起来，就跟 map() 函数的功能类似。它开始的语法就像这样 x <- myCollection，这跟 Java 5 里的 for 循环增强版很像。紧接着是一个可选的像 filter() 一样的守卫。最后是 yield {} 结构，它实际上有点像传递给 map() 的参数，不同的是传递给 map() 的函数要在 for 循环前面声明。

竞赛要求输出特定格式的预测结果。社交圈子用空格分隔的用户 ID 列表表示，并且每个社交圈子间用分号（;）分隔。代码如下：



在上面的示例中，我们已按照竞赛要求的格式输出用户 ID 和社交圈子结果数据，但是最终输出还要依赖于随后这些数据放到哪里。Spark 允许方便地把数据推送到外部数据库、实时系统或集成到更复杂的机器学习 pipeline 中。

5.5 受欢迎的回馈：增强连通组件

对于有向图，有时我们可能要消除组件中闭塞不通的死胡同路径，如图 5.10 所示。在社交网络中，如果其他方面的算法被添加到推荐引擎中，增强连通组件会是构成推荐引擎的基础。另一个应用是确保在一个状态机中没有闭塞不通的死胡同，如果状态机中有则会被卡住。当编译器做数据流分析来识别从来没有被用到的表达式时，增强连通组件在构建优化编译器时也很有用，否则会浪费计算资源。

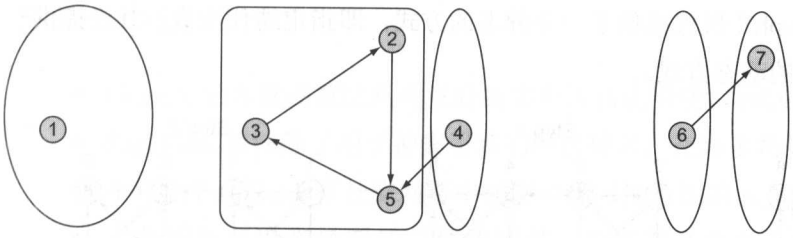


图 5.10 在强连通组件中，每个顶点都可以从其他顶点到达，没有闭塞不通的死胡同路径。

调用 stronglyConnectedComponents() 和调用 connectedComponents() 很相似，唯一不同的是，stronglyConnectedComponents() 要求传入 numIter 参数。假设图对象 g 已经构建好了，清单 5.6 的作用就是找到其中的强连通组件。

清单 5.6 调用强连通组件方法 stronglyConnectedComponents()

```
g.stronglyConnectedComponents(10).vertices.map(_.swap).groupByKey.
    map(_._2).collect
res4: Array[Iterable[org.apache.spark.graphx.VertexId]] = Array(
CompactBuffer(4), CompactBuffer(1), CompactBuffer(6), CompactBuffer(7),
CompactBuffer(3, 5, 2))
```

5.6 社区发现算法：标签传播

为了识别出图中紧密交织的群体，GraphX 提供了标签传播算法（LPA），这个算法由拉加万等人在 2007 年的论文《近似线性时间算法在大规模网络中检测社区结构》中提出。这个想法是让稠密连接的顶点组在一个唯一的标签上达成一致，所以这些顶点组被定义为一个社区。

> **定义**：如果迭代算法在每轮迭代中都能保证接近一个特定结果，这时就可以说这个算法是“收敛”的。对于有收敛特性的算法，运行算法要求尽可能多的迭代次数，且当两次迭代结果足够接近时使用一个公差来判断是否退出迭代，上述两种情况都是合理的。不收敛的算法会永远运行下去，所以我们需要指定一个迭代次数上限。不可避免地需要在最终结果的精确度和算法耗时上做权衡。

不幸的是，LPA 常常不是收敛的。如图 5.11 所示，这是一个不收敛的例子，图在第 5 步和第 3 步的结果相同，算法将会在与第 4 步和第 5 步之间永远循环下去。

对于这种情况，GraphX 仅仅提供了一个静态的方式，即指定迭代次数，并没提供一个带公差终止条件的动态方式。

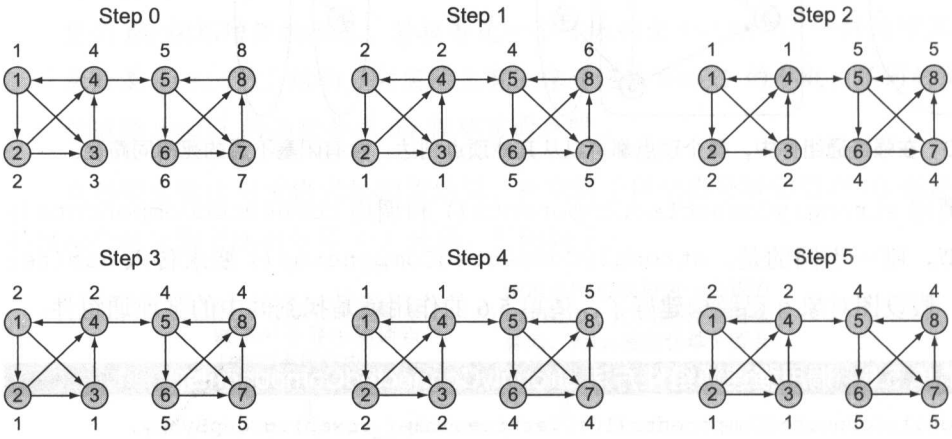


图 5.11 LPA 算法常常不收敛。第 5 步和第 3 步结果相同，意味着在第 4 步和第 5 步之间永远地重复循环下去。

尽管从 LPA 的名字上听起来可以做标签传播，实际上 LPA 不适合传播顶点分类的情况，即从已知分类的顶点向未知分类的顶点传播标签。7.3 节会探讨这种情况，将之称为半监督学习。

相比之下，LPA 在图 5.11 所示的第 0 步对顶点 ID 的标签进行初始化（见清单 5.7）。LPA 并不关心边的方向，实际上是把图当作无向图来用。正如图 5.11 中第 3 步和第 5 步中两个来回反转的标签集与拉加万的论文中的简单示例类似。

清单 5.7 使用标签传播

```
val v = sc.makeRDD(Array((1L,""), (2L,""), (3L,""), (4L,""), (5L,""),
 (6L,""), (7L,""), (8L,"")))
val e = sc.makeRDD(Array(Edge(1L,2L,""), Edge(2L,3L,""), Edge(3L,4L,""),
 Edge(4L,1L,""), Edge(1L,3L,""), Edge(2L,4L,""), Edge(4L,5L,""),
 Edge(5L,6L,""), Edge(6L,7L,""), Edge(7L,8L,""), Edge(8L,5L,""),
 Edge(5L,7L,""), Edge(6L,8L,"")))
lib.LabelPropagation.run(Graph(v,e),5).vertices.collect.
 sortWith(_._1<_._1)
res5: Array[(org.apache.spark.graphx.VertexId,
     org.apache.spark.graphx.VertexId)] = Array((1,2), (2,1), (3,1), (4,2),
     (5,4), (6,5), (7,5), (8,4))
```

5.7 小结

- GraphX 的内置图算法具有很好的实用性和适用性，功能强大，应用范围广泛。
- PageRank 算法除了用于搜索引擎的网页排名，在许多场景中都非常有用。
- 个性化 PageRank 在社交网络中判断“你可能认识的人”时非常有用。
- 三角形数可以衡量图的总体连通性，而在第 8 章引入的另外一个度量方式——全局聚集系数的优点是让结果总是在 0 到 1 这个指定的范围中，可以方便比较不同大小的图。
- 连通组件和增强连通组件可以在社交网络中找出社交圈子。
- GraphX 的标签传播算法没什么大用，因为它很难收敛。

6 其他有用的图算法

本章要点

- GraphX 之外的标准图算法
- 有权值的最短路径图算法
- 旅行推销员问题（TSP）
- 最小生成树

第 5 章介绍了基本的 GraphX API，现在就可以根据实际需要写一些自定义的图算法了。有了这些 GraphX API 提供的标准图算法实现，也就没必要重新造轮子了。而有一些算法，数十年来与图有关联而 GraphX 没有提供，这一章就介绍这类经典的图算法，探讨一下哪些还适合使用。

这些经典的图算法始于 1950 年，远远早于 Spark 和其他的并行计算技术，它们本质上是迭代算法，比如每次迭代添加一条边。GraphX 的 Pregel API 由于是同时操作所有顶点，所以不是很适合。尽管在算法迭代过程中每一步都会调用一些图的广度搜索，但是 GraphX 的并行处理能力仍然会被用到。下面你会看到用 GraphX 的迭代功能 Map/Reduce（函数 `aggregateMessages()` 和函数 `outerJoinVertices()`）来实现和并行化这些原本被设计为顺序执行的算法。

在本章将要介绍的三个算法中，第一个是有权值的最短路径算法，填补了GraphX API一个明显的缺失，GraphX中只提供了每条边权值为1的最短路径算法。带权值的最短路径允许在路由映射表上规划，其中每条边的权值表示两个顶点（比如城市）的距离。

第二个算法是旅行推销员问题（TSP），在图中找到一条访问每个顶点一次并回到起点的最短路径。这个算法在包裹或邮件传递以及物流应用方面非常有用。

第三个算法是最小生成树算法，在一棵树（无环图）中，找到一个生成树，其边权值之和小于任何其他生成树边权值之和。听起来有些抽象（其实也是前面提到的有权值的最短路径图算法之一），在路由效用方面很有用，也有一些非直观的使用，如创建科学文献的分类或分级。[译注1]

6.1 你自己的GPS：有权值的最短路径

今天，我们认为在智能手机和地图应用中有GPS全球定位系统功能是理所当然的。但如何选择最佳路径呢？Edsger Dijkstra在1956想通了这个问题，提出了最短路径算法，这个算法已经在Spark中实现。

5.3节讲了GraphX实现的无权值的最短路径图算法，而Dijkstra的算法是在图中用有权值的边查找最短路径（参见图6.1）。

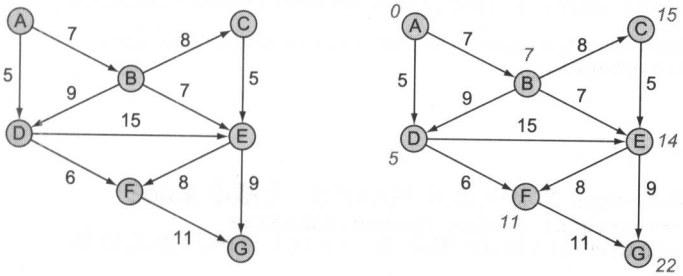


图6.1 示例图数据，以及运行Dijkstra算法后距顶点A的距离。左图是给定一个边有权值的图，Dijkstra算法用“从顶点A到当前顶点的最短距离”标注每个顶点。图数据来源：图的数据来自关于公开的Kruskal算法的维基百科文章（顺便说一句，在本章的最后一部分实现了Kruskal算法）。

译注1 一个连通图可能有多个生成树。当图中的边具有权值时，总会有一个生成树的边的权值之和小于或者等于其他生成树的边的权值之和。广义上而言，对于非连通无向图来说，它的每一个连通分量同样有最小生成树。——来自wikipedia.org

Dijkstra 算法是在图中计算从一个指定的顶点到其他每一个顶点的路径距离。可以这样描述 Dijkstra 算法：

1. 初始化起始顶点为距离 0，到其他顶点的距离为无穷大。
2. 设置当前顶点为起始顶点。
3. 在当前顶点的所有相邻顶点上，当前顶点的距离加上从当前顶点连向其他顶点的边长度之和，这二者中取小的值设置为当前顶点的距离值。例如在图 6.1 中，第一次迭代计算后，顶点 D 的最短路径值是 5，顶点 B 的最短路径值是 7；第二次迭代计算后，多了一条可选路径 A->B->D，A 到 D 的总路径长度是 16，所以选择 D 的第一次迭代计算出的最短路径值 5。
4. 标记当前节点为已访问。
5. 设置有最短路径值的顶点为当前顶点，并设为未访问顶点。如果已经访问过所有节点，则停止本次迭代。
6. 跳转到第 3 步。

Dijkstra 算法有许多变种，包括有向图和无向图的版本。清单 6.1 实现的是针对有向图的。

清单 6.1 Dijstra 最短路径算法[译注 2]

```
def dijkstra[VD](g: Graph[VD, Double], origin: VertexId): Graph[(VD,
Double), Double] = {
    import org.apache.spark.graphx._
    //  1. 初始化
    var g2 = g.mapVertices {
      case (vid, _) =>
        val vd = if (vid == origin) 0 else Double.MaxValue
        (false, vd)
    }

    //  2. 遍历所有的点
    (0L until g.vertices.count).foreach { i: Long =>

    //  3. 确定最短路径值最小的作为当前顶点
```

译注 2 译者做了简单的代码重构格式化，以方便代码阅读。

```
    val currentVertexId: VertexId = g2.vertices.filter(!_._2._1)
      .fold((0L, (false, Double.MaxValue))) {
        case (a, b) => if (a._2._2 < b._2._2) a else b
      }._1

    // 4. 向与当前顶点相邻的顶点发消息，再聚合消息：取小值作为最短路径值
    val newDistances: VertexRDD[Double] = g2.aggregateMessages[Double](
      // sendMsg: 向邻边发消息，内容为边的距离与最短路径值之和
        ctx => if (ctx.srcId == currentVertexId) ctx.sendToDst(ctx.
srcAttr._2 + ctx.attr),
      // mergeMsg: 选择较小的值为当前顶点的相邻顶点的最短路径值
      (a, b) => math.min(a, b)
    )

    // 5. 生成结果图
    g2 = g2.outerJoinVertices(newDistances) { (vid, vd, newSum) =>
        (vd._1 || vid == currentVertexId, math.min(vd._2, newSum.
getOrElse(Double.MaxValue)))
    }
  }

  g.outerJoinVertices(g2.vertices) { (vid, vd, dist) =>
    (vd, dist.getOrElse((false, Double.MaxValue))._2)
  }
}
```

Spark 小贴士：RDD API 没有如标准 Scala 集合类中的 minBy() 函数，所以上述代码用 fold() 来实现 minBy() 的功能。

Dijkstra 最短路径算法的实现，用了 var 变量（即 g2）而不是 val 变量，这是因为迭代计算算法要把每一轮迭代计算的结果赋值给一个变量以便在下一轮迭代中使用。首先初始化变量 g2，去掉原来的图 g 的顶点数据，添加由 Boolean 和 Double 组成的键值对。Boolean 变量表示顶点是否被访问过，Double 表示源顶点到当前顶点的距离。

截止到 GraphX 1.6 版本，所有的图都是不可变的，在这些图算法中要更新图中

的顶点的属性值只能创建一个新图。当计算出新距离值时，调用 outerJoinVertices() 函数更新距离值生成一个新图，赋值给 g2 变量，g2 变量被重新赋值前的引用对象不再被使用，Java 虚拟机会在适当时候回收这个老的图对象。

代码中的最后一行返回值，用 outerJoinVertices() 函数把新图 g2 的顶点属性重新保存到原来的图 g 中。这一步，return 所返回的图的顶点属性被更新，顶点属性类型是 Tuple2[VD,Double]，VD 类型不变，Double 类型对应的值被 Dijkstra 算法计算出的新距离值替换。

Pregel API 由于有“当前顶点”的概念而不容易使用，即当前迭代中全局全部最小的顶点。Pregel API 更适合于视每个顶点都对等的算法。下面就来看一下如何在图 6.1 所示的图上执行新的 dijkstra() 函数。

清单 6.2 执行最短路径距离算法

```
val myVertices = sc.makeRDD(Array((1L, "A"), (2L, "B"), (3L, "C"),
    (4L, "D"), (5L, "E"), (6L, "F"), (7L, "G")))
val myEdges = sc.makeRDD(Array(Edge(1L, 2L, 7.0), Edge(1L, 4L, 5.0),
    Edge(2L, 3L, 8.0), Edge(2L, 4L, 9.0), Edge(2L, 5L, 7.0),
    Edge(3L, 5L, 5.0), Edge(4L, 5L, 15.0), Edge(4L, 6L, 6.0),
    Edge(5L, 6L, 8.0), Edge(5L, 7L, 9.0), Edge(6L, 7L, 11.0)))
val myGraph = Graph(myVertices, myEdges)

dijkstra(myGraph, 1L).vertices.map(_._2).collect

res0: Array[(String, Double)] = Array((D,5.0), (A,0.0), (F,11.0), (C,15.0),
  (G,22.0), (E,14.0), (B,7.0))
```

这是图 6.1 的计算结果，但它是怎么通过路径顺序来得到这些目标顶点的呢？其实，这个算法是计算距离而不是路径。

清单 6.3 列出的代码添加了一个顶点属性对象 List[VertexId] 来记录每一次迭代时的路径。

清单 6.3 包含路径记录的 Dijkstra 最短路径算法

```
  def dijkstraTrace[VD](g: Graph[VD, Double], origin: VertexId) = {
    import org.apache.spark.graphx._
    var g2 = g.mapVertices((vid, vd) => (false, if (vid == origin) 0 else
Double.MaxValue, List[VertexId]()))
```

```
    for (i <- 1L to g.vertices.count - 1) {
      val currentVertexId =
       g2.vertices.filter(!_._2._1).fold((0L, (false, Double.MaxValue,
List[VertexId]()))) {
        (a, b) => if (a._2._2 < b._2._2) a else b
      }._1

       val newDistances: VertexRDD[(Double, List[VertexId])] = g2.
aggregateMessages[(Double, List[VertexId])](
          ctx => if (ctx.srcId == currentVertexId) ctx.sendToDst((ctx.
srcAttr._2 + ctx.attr, ctx.srcAttr._3 :+ ctx.srcId)),
        (a, b) => if (a._1 < b._1) a else b
      )

      g2 = g2.outerJoinVertices(newDistances)((vid, vd, newSum) => {
       val newSumVal = newSum.getOrElse((Double.MaxValue, List[VertexId]()))
       (vd._1 || vid == currentVertexId, math.min(vd._2, newSumVal._1), if
(vd._2 < newSumVal._1) vd._3 else newSumVal._2)
      })
    }

    g.outerJoinVertices(g2.vertices) { (vid, vd, dist) =>
      (vd, dist.getOrElse((false, Double.MaxValue, List[VertexId]())))
    }
  }
```

Scala 小贴士：运算符（在 Scala 里，这些符号看起来像个运算符，其实是一个函数）：+ 是 Scala List 里的一个函数，返回追加了一个元素的新集合列表。Scala List 里有不少类似这样的在元素或集合列表前追加或在集合列表后追加的运算符，这些在 Scaladocs 中的 List 介绍里都有。

清单 6.4 执行包含路径记录的 Dijkstra 最短路径算法

```
dijkstra(myGraph, 1L).vertices.map(_._2).collect
res1: Array[(String, List[Any])] = Array((D,List(5.0, List(1))),
 (A,List(0.0, List())), (F,List(11.0, List(1, 4))),
```

```
(C,List(15.0, List(1, 2))), (G,List(22.0, List(1, 4, 6))),
(E,List(14.0, List(1, 2))), (B,List(7.0, List(1))))
```

这样看起来就好多了，能清晰地看到到任何其他顶点的最短路径。

6.2 旅行推销员问题：贪心算法

旅行推销员问题（TSP）是在一个无向图中找到一个经过每一个顶点的最短路径。假如有一个推销员，他要到某一地区的所有城市去推销，他想要走过的总路程最少。

与上一小节的最短路径算法不同，没有一个简单直接的确定性算法来解决这个旅行推销员问题。旅行推销员问题是一个众所周知的数学问题，这个新术语始于 1930 年，这里的这个术语特指这个数学问题，并不是字面上的意思。

这个问题是一类被称为 NP-hard（non-deterministic polynomial-time hard）的问题，意味着在一个时间量里它不能被解决，相对于顶点数或边数这就是一个多项式。相反，一个组合优化问题要求越来越快地求最优解。不同于尝试找到最优解，有很多近似解接近最优解。图 6.2 所示的实现用的是贪心算法（greedy algorithm），这虽然是最简单的算法，但有可能会给出偏离最优解的答案，并且不一定会到达所有的顶点（如果要求到达每个顶点，算法可能会给出不可接受的答案）。之所以称为贪心算法，是因为在每一次迭代中它会选择最接近的最短边，而没有做任何进一步搜索。

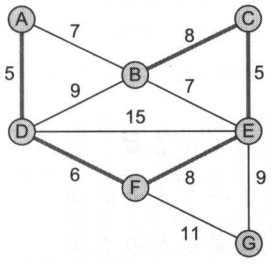


图 6.2 对于旅行推销员问题来说，贪心算法是最简单的，缺点是不会总是到达所有顶点。在这个例子中，顶点 G 就没有到达。

贪心算法可在不用增加太多代码的情况下，用不同的起始顶点重新运行整个算法，不断迭代，挑选出一个到达所有顶点并且最短的解决方案。用这种方法就可以改善贪心算法的效果，代码如清单 6.5 所示。

贪心算法比较简单，步骤如下：

1 从某些顶点开始。

2 添加权重最小的邻边到生成树。

3 跳到第 2 步。

清单 6.5 旅行推销员问题的贪心算法实现

```
def greedy[VD](g: Graph[VD, Double], origin: VertexId) = {
    var g2: Graph[Boolean, (Double, Boolean)] = g.mapVertices((vid, vd) =>
vid == origin).mapTriplets {
      et => (et.attr, false)
    }
    var nextVertexId = origin
    var edgesAreAvailable = true
    type tripletType = EdgeTriplet[Boolean, (Double, Boolean)]
    do {
      val availableEdges = g2.triplets
                            .filter { et => !et.attr._2
        && (et.srcId == nextVertexId && !et.dstAttr
            || et.dstId == nextVertexId && !et.srcAttr)
      }
      edgesAreAvailable = availableEdges.count > 0
      if (edgesAreAvailable) {
        val smallestEdge = availableEdges.min()(new Ordering[tripletType]() {
          override def compare(a: tripletType, b: tripletType) = {
            Ordering[Double].compare(a.attr._1, b.attr._1)
          }
        })
        nextVertexId = Seq(smallestEdge.srcId, smallestEdge.dstId)
                       .filter(_ != nextVertexId).head
        g2 = g2.mapVertices((vid, vd) => vd || vid == nextVertexId)
               .mapTriplets { et => (et.attr._1, et.attr._2 ||
                   (et.srcId == smallestEdge.srcId
                    && et. dstId == smallestEdge.dstId))
        }
      }
```

```
    } while (edgesAreAvailable)

    g2
}
```
译注3

Scala小贴士：type是Scala的保留字，用来对复杂的类型做别名，以防在编译期一遍又一遍地声明很长的类型。type与C/C++的类型定义相似，而不同于Java中的类型。type的另一个用处是在Scala中引入抽象类型成员，这方面的细节不在本书讨论范围内，读者可以参考scala-lang.org的官方文档。

在清单6.5中，声明了三个变量。其中一个变量g2用在do-while循环控制语句里，这里用变量是因为Scala里没有break关键字（虽然标准类库里也有一个break模式）。在循环里，与函数传入的参数对象g相比，对象g2有不同的顶点属性类型和边属性类型。g2的顶点属性是Boolean类型，标示这个顶点是否已被最终方案采纳；边属性是Tuple2[Double,Boolean]类型，Double是算法中用到的边权重值，Boolean表示这条边是否被采纳到最终方案里了。

availableEdges这个变量用于在来源和目标两个方向上检查边的有效性，也就是说，贪心算法用到的图是无向图。所有的GraphX图实际上都是有向图，有来源和目标两个方向，所以任何基于GraphX的无向图算法都要自己在来源和目标方向上做一些预先检查。

在清单6.5中，类似于上一小节中的最短路径算法，由于GraphX中的图都是不可变的，在算法迭代过程中，要对边和顶点的Boolean属性重新设置一个新Boolean值，图有了改变，所以就要创建一个新的图赋值给g2变量，以便在下一轮迭代中继续使用。还有一点也与最短路径算法类似，Pregel API在这种场景下不是一个好的选择，这是因为解决旅行推销员问题的贪心算法会依次添加一条边（到一个顶点上）；这就不是视所有的顶点对等了。

贪心算法最后生成的结果图对调用方来说可能不是最方便的数据格式，但也不会退回使用原始图g的顶点属性数据格式，因为最终图是连接了图g和图g2的，并且GraphX也没有提供自动化连接图的方法。下一节我们会用几行Scala代码实现这个功能。

译注3 译者做了简单的代码重构格式化，以方便阅读。

6.3 路径规划工具：最小生成树

最小生成树听起来比较抽象，貌似没什么用处。我们可以像对待旅行推销员问题一样，除非需要否则不用关心其内部原理。最小生成树的最直接应用是在路径规划工具方面（道路、电力、水等），用来确保这些基础设施资源能在最小消耗的前提下到达所有城市（例如最短距离，路径图的边权值表示城市间的距离）。也有一些不太显著的应用，如在相似事物的集合上做分类，例如动物（用于科学分类）或报纸头条。示意图如图 6.3 所示。

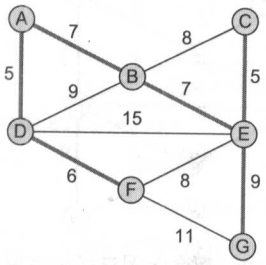


图 6.3 最小生成树是一棵树（无环图），也是一个到达所有顶点的无向图，具有最小总权值（边的权值总和）。

清单 6.6 是 Kruskal 算法的实现。本章用到的例图和维基百科上的 Kruskal 算法的例图一样，通过一系列的图说明了算法的执行过程，这样我们就可以清楚地看到算法实现的原理。

虽然 Kruskal 算法也是一种贪心算法，但它的确能从多个最小生成树中查找出一个最优解（因为可能会有多个最小生成树有相同的总权值）。查找出一个最小生成树未必就是组合问题。Kruskal 算法之所以是贪心算法，是因为在每一次迭代计算过程中它都会快速查找到具有最小权值的边。不像旅行推销员问题那样最终结果是一个可以用数学证明的最小生成树。示意图如图 6.4 所示。

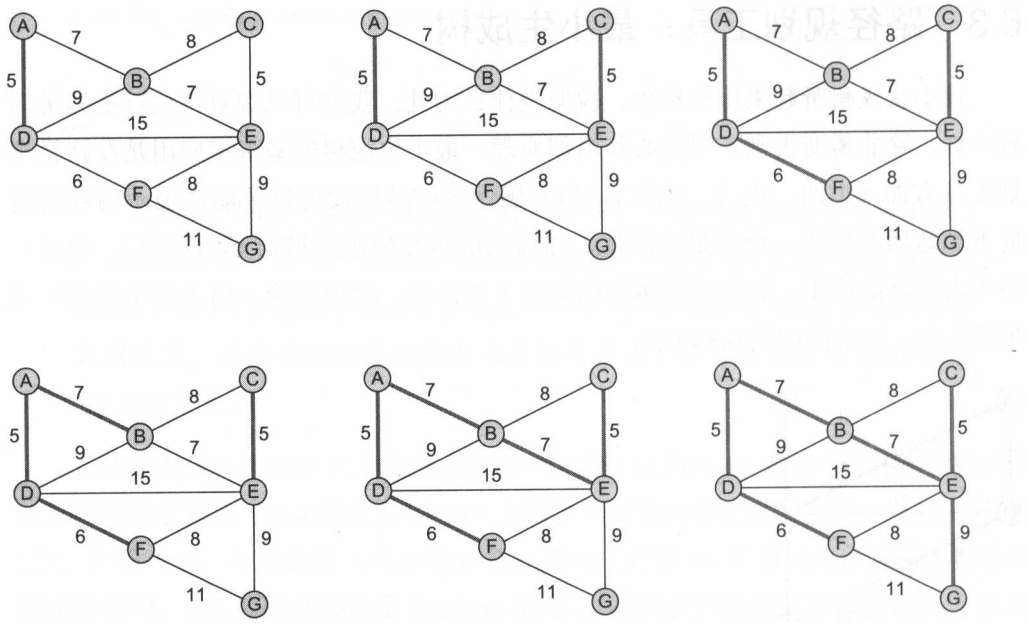


图 6.4 查找最小生成树的 Kruskal 算法的迭代计算过程。在每一次迭代中，都会全图查找没有用到的边的最小权值。但要注意：边不能构成一个环（正在被查找的树）。

与上一节中的旅行推销员算法不同，Kruskal 算法不能遍历一次边就把最终结果树构建出来。相反，它做一个全图搜索来发现具有最小权值的边，将它添加到边的集合中，最终构建出一棵树。算法描述如下：

1 初始化集合中的边，构建一个空的最小生成树。

2 通过全图查找找到具有最小权值的边，符合以下两个条件就将其添加到结果集合中。

- 边不在结果集合中。
- 边不能与结果集合中的边构成环。

3 回到第 2 步，直到所有的顶点都包含在边的结果集中。

第 2 步的第二个条件是一个难点。很容易理解如何查找一个环，却没有一个明确直接的方法描述给计算机。不过也有一些间接方法。为每一个候选边（开始时，全部或者大部分的边）找到最短路径（例如，通过调用 GraphX 的内置的 `org.apache.spark.graphx.lib.ShortestPaths`，见 5.3 节），丢弃这些顶点已经与其他边有连接的边。另一个方法在清单 6.6 中用到了，即 5.4 节提到的调用 GraphX 的内置函数 `connectedComponents()`。这个方法一下子通过全图给出全部顶点的

连接信息。如果一条边的两个顶点属于相同结果树的连接分量，对 Kruskal 算法来说这个边就不符合要求，因为添加到结果边集合中会生成环。

清单 6.6 最小生成树

```
def minSpanningTree[VD: scala.reflect.ClassTag](g: Graph[VD, Double]) = {
  var g2 = g.mapEdges(e => (e.attr, false))
  for (i <- 1L to g.vertices.count - 1) {
    val unavailableEdges =
      g2.outerJoinVertices(g2.subgraph(_.attr._2)
                           .connectedComponents
                           .vertices)((vid, vd, cid) => (vd, cid))
        .subgraph(et => et.srcAttr._2.getOrElse(-1) == et.dstAttr._2.
getOrElse(-2))
        .edges
        .map(e => ((e.srcId, e.dstId), e.attr))
    type edgeType = ((VertexId, VertexId), Double)
    val smallestEdge =
      g2.edges
        .map(e => ((e.srcId, e.dstId), e.attr))
        .leftOuterJoin(unavailableEdges)
        .filter(x => !x._2._1._2 && x._2._2.isEmpty)
        .map(x => (x._1, x._2._1._1))
        .min()(new Ordering[edgeType]() {
          override def compare(a: edgeType, b: edgeType) = {
            val r = Ordering[Double].compare(a._2, b._2)
            if (r == 0)
              Ordering[Long].compare(a._1._1, b._1._1)
            else
              r
          }
        })
    g2 = g2.mapTriplets(et =>
      (et.attr._1, et.attr._2 || (et.srcId == smallestEdge._1._1
                                  && et.dstId == smallestEdge._1._2)))
  }
  g2.subgraph(_.attr._2).mapEdges(_.attr._1)
```

```
}
//   运行函数
 val retMST = minSpanningTree(myGraph).triplets.map(et => (et.srcAttr,
et.dstAttr)).collect
println(retMST.mkString(", "))
// 输出：(A,B), (A,D), (B,E), (C,E), (D,F), (E,G)
```

Scala 小贴士： 在用到 Scala 范型时，有时候声明类型参数 scala.reflect.ClassTag 也是必要的，因为 JVM 在运行期有类型擦除。在清单 6.6 中，在 JVM 编译期和运行时，调用 subgraph() 时都需要类型 VD 的声明。

在上述代码实现中，迭代过程中唯一的类型变化是添加了一个 Boolean 值到 Edge 边属性中，来标示这条边是否是生成树的最终边结果集合的一部分。

巧妙的地方是赋值给变量 unavailableedges。首先在图 g2 中筛选出已经在结果集中存在的边，构造为一个子图，然后这个子图执行 connectedComponents() 函数，通过计算每一个顶点连通分量关系，返回一个包含最小顶点 ID 的图，获取这个图的顶点集合，然后与使用 outerJoinVertices 图 g2 做连接。这样就可以判定，一个边的两个顶点属于相同的连通分量，那么这个边就不符合条件。根据 outerJoinVertices 的功能，这些分量 ID 可以是 null（Scala 里用 Option[VertexId] 表示，None 值表示 null 的含义）。如果一个顶点不属于增长中的边的结果集合，接下来就轻松了，通过判断默认 ID 值为 -1 的连通分量 ID 和默认 ID 值为 -2 的连通分量 ID 是否相等，来防止这样的边被声明为不可用的。故意选择 -1、-2 不仅因为这是无效的顶点 ID，也因为包含两个彼此不同顶点的尚且不在结果集中的边会被认为仍然有效。

smallestEdge 变量所引用的计算过程（函数）主要是边连接的概念，本章也多处提到了 GraphX 没有内置实现的边连接。map() 函数把边转化成 Tuple2[Tuple2[VertexId, VertexId], Tuple2[Double, Boolean]] 类型，然后调用 RDD 标准的左外连接函数 leftOuterJoin（不要与 GraphX 标准的 join() 和 outerJoin() 函数混淆）。leftOuterJoin() 会视 Tuple2[VertexId, VertexId] 为一个独立的对象实体，在执行连接操作时会分开 Tuple2 中的两个顶点 ID 到不同的对象中（见 GraphX 中 leftOuterJoin() 函数的定义：def leftOuterJoin[W](other: RDD[(K, W)],partitioner: Partitioner):

RDD[(K, (V, Option[W]))])。执行完 leftOuterJoin()，结果数据就会分为如图 6.5 所示的 5 个部分。

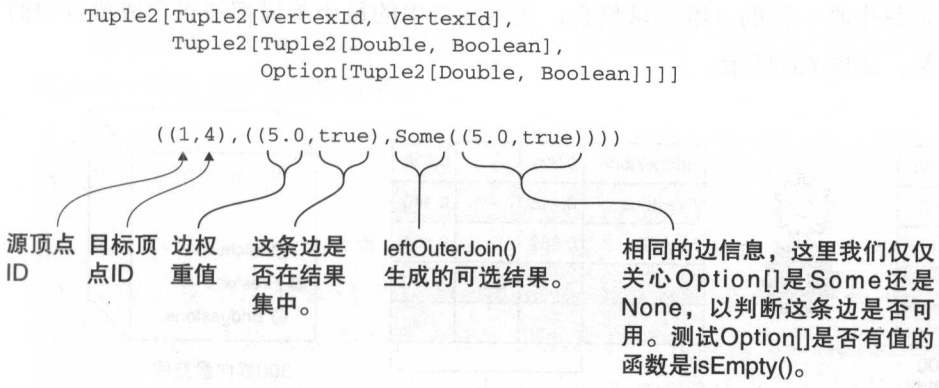


图 6.5 清单 6.6 中的 leftOuterJoin() 执行以后返回的结果数据类型。

在去除不可用的边之后，map() 随后提取我们关心的两条信息：VertexId 对和边的权重。override def compare 首先比较边的权重，如果相等，则通过比较看哪个 VertexId 小。这是为了使执行具有确定性和可重复性，并匹配维基百科示例中的结果，这显然也是在权重相同时如何取舍的做法。如果你不关心可重复性或匹配维基百科（即不关心尝试确保你的程序匹配那个已知的结果），可以用 Ordering[Double].compare(a._2, b._2) 替换 override def compare 的方法体。

整个算法实现函数 minSpanningTree() 的返回值就是树本身，不是全图。边属性类型（Edge 的属性）被恢复成权值，临时的 Boolean 值被丢弃。

6.3.1 基于 Word2Vec 的推导分类法和最小生成树

一个研究最小生成树的方法是把它们当成提取（某种意义上）Graph 图中最重要的连接。通过剔除一些不重要的边让图变得稀疏，减少到只有最基本的必需元素。本节将展示如何用机器学习和图处理来将一个简单的不连接的动物名称列表变成一个有连接的用到最小生成树（MST）的分类。

最小生成树不是万金油，自然语言处理工具 Word2Vec 能弥补最小生成树算法的一些不足。Word2Vec 可以在每个关系中分配一个距离数值，以至于可以在关系上构建一个有权重的连通图，然后运行最小生成树的图算法来发现最重要的连接关系。

理解 Word2Vec

Word2Vec 是一个自然语言处理算法，它把一个文本集（文本文档集合）转换成一个表示每个唯一词的 *n* 维向量集合。文本集合中的每个单词都被表示成集合中的一个向量，如图 6.6 所示。

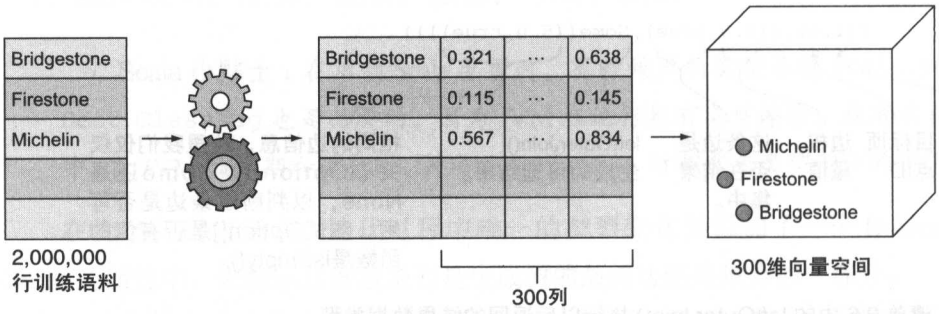


图 6.6 从训练数据集中抽取出词，用 Word2Vec 算法根据这些词生成每一个词对应的 n 维向量，这里的 n 是 300。语义相似的词如 Michelin 和 Firestone 的值就比较相近。

Word2Vec 生成的向量集合的用处是，语义上相似的词往往是紧密联系在一起的。我们可以用一个称为*余弦相似度*的相似性度量方法计算出这些词的相似度。余弦相似度的取值范围是从 1（完全相同）、0（不相似，互相独立）到 -1（完全不同）。然后用 1 减去余弦相似度得到余弦距离。

余弦距离 = 1 - 余弦相似度

现在我们有了这个度量方法，相似的词有较小的距离差值，差值大的词就表示它们不相似。

用到 Word2Vec 的训练模型往往需要大量的有效数据。幸运的是已经有许多好的预训练模型可以用，但是这样的训练任务都是用基于 Google 新闻（1000 亿字）的子数据集的训练模型。虽然 Spark 的机器学习类库 MLlib 也有 Word2Vec 的实现，但是它没有加载二进制的 Google 新闻数据模型的功能。相反，我们已经预先计算出了动物列表数据中的每一个名字的余弦距离，把这些数据保存在一个用逗号分隔的名为 animal_distances.txt 的文件中，这个文件包含在本章的代码下载文件中。如果这个文件中有如下数据：

```
sea_otter,sea_otter,-0.000000
sea_otter,animal,0.638965
sea_otter,chicken,0.860217
```

```
sea_otter,dog,0.705229
sea_otter,aardvark,0.767667
sea_otter,albatross,0.770162
```

这个数据文件每行用逗号分隔成三列，每行包含一对关系和它们的余弦距离。列表中大概有 224 个动物名字。

创建余弦距离文件

因为 Spark MLlib 不能加载其他语言实现的 Word2Vec 算法所生成的 Word2Vec 模型，那么这里就用 Python 类库 Gensim 来加载。如果你想自己用 Word2Vec 算法生成余弦距离文件，那就按照页面 https://radimrehurek.com/gensim/install.html 上的安装指令进行操作。一般需要用到 easy_install：

```
easy_install -U gensim
```

或者 pip：

```
pip install --upgrade gensim
```

然后从 https://code.google.com/archive/p/word2vec/ 下载 Google 新闻模型。这个文件有几个 GB 大小。

现在准备生成余弦距离，用 animal_terms.txt 文件中的 200 个动物关系列表，代码如下所示：

```
from gensim.models import Word2Vec
model = Word2Vec.load_word2vec_format
        ('GoogleNews-vectors-negative300.bin', binary=True)
f = open('animal_terms.txt')
animals = f.read().splitlines()
animals = [x.lower() for x in animals if x.lower() in
           model.vocab.keys()]
f.close()
f = open('animal_distances.txt','w')
f.truncate()
for i in range(0, len(animals)):
    for j in range(i, len(animals)):
        f.write( '%s,%s,%1.6f' %
```

```
            (animals[i], animals[j],
            1 - model.similarity(animals[i], animals[j])))
f.flush()
f.close()
```

输出结果是 animal_distances.txt 文件，它用于构建包含距离属性的图。

创建最小生成树

现在我们用上面的数据列表构建一个基于距离的动物连接关系图，距离来自于 Google 新闻 Word2Vec 模型。

定义：一个完整的图是每一个顶点都有一个和其他顶点连接的边。图中的边数 e 和顶点数 v 之间的函数关系是：$e = n(n-1)/2$。

每一个动物都是一个顶点，动物间的连接是有权值的边，边的权值来自于动物向量中的余弦距离。由于为每两个动物都生成一条边，所以图是完整的。

清单 6.7 展示了如何构建图并生成最小生成树。用第 4 章开发的 `toGexF()` 函数持久化生成树到文件中，可以读取这个结果文件并用 Gephi 可视化展示。用 200 多个顶点生成的树已经很大了，在图 6.7 中展示了一部分。

清单 6.7 构建距离图

```
val dist = sc.textFile("animal_distances.txt")
val verts = dist.map(_.split(",")(0)).distinct.
    map(x => (x.hashCode.toLong,x))
val edges = dist.map(x => x.split(",")).
    map(x => Edge(x(0).hashCode.toLong,
                  x(1).hashCode.toLong,
                  x(2).toDouble))
val distg = Graph(verts, edges)
val mst = minSpanningTree(distg)
val pw = new java.io.PrintWriter("animal_taxonomy.gexf")
pw.write(toGexf(mst))
pw.close
```

animal_distances.txt文件中的前两列包含动物名，选择第一列，应用距离函数来保证顶点唯一。

调用hashCode，并转成Long值，这个Long值作为每个顶点的VertexID。

调用最小生成树算法。

这个例子展示了如何获得准确的一组关系之间的语义连接。我们举了动物名称的例子，类似的方法也可以用于其他有大量非结构化文本数据的领域，如医学文献或全球证券交易所上市公司的报告。

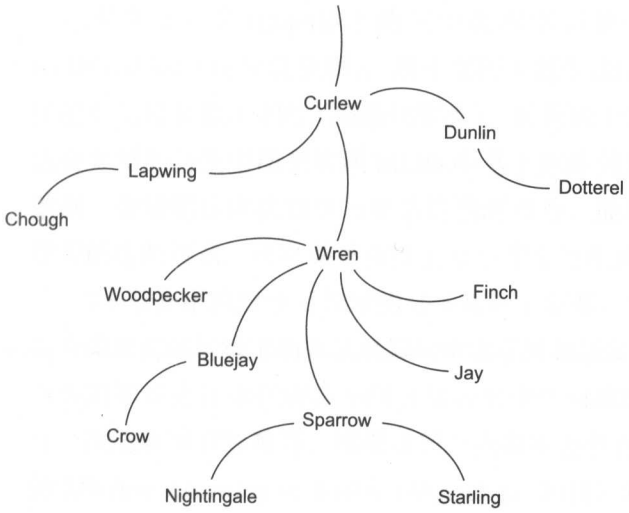


图 6.7 部分动物分类的最小生成树，示例展示了鸟之间的关系连接。

6.4 小结

- 许多经典的图算法没有用 Pregel 实现。我们学习了带权值的最短路径、旅行推销员问题、最小生成树的自定义实现。
- 最小生成树的裁剪图，缩减成只有必需的边。
- Spark 可以轻松地把图处理和其他机器学习算法结合起来，就像前面看到的动物分类的例子。

7 机器学习

本章要点

- 基于图的机器学习
- 监督学习：电影推荐系统，垃圾信息检测
- 无监督学习：文档聚类，基于聚类的图像分割
- 半监督学习：使用向量生成图
- 在 GraphX 中使用 Spark MLlib

机器学习是广义人工智能（AI）下的一个子领域，在已知一部分参考数据的情况下进行数据的预测，例如基于用户喜欢的《星球大战》来预测用户是否也喜欢《帝国反击战》。

尽管机器学习是 AI 的一个子领域，但它依然包含了相当丰富的内容。机器学习里有几十个不同的算法种类和几百个不同的标准算法，囊括了形形色色的使用案例和各不相同的实现思路。很多算法使用矩阵作为它们主要的数据结构，但也有不少是使用图的。Spark MLlib 主要关注的就是基于矩阵的机器学习算法，当然它也有部分算法是使用 GraphX 进行处理。MLlib 和 GraphX 也会有重叠的部分：GraphX 包括了用于推荐系统的 SVDPlusPlus 算法。

本章节包含 GraphX 上的三个机器学习算法（Spark 1.6）：用于商品推荐的 SVDPlusPlus（奇异值分解），用于文档主题生成的 LDA（隐含狄利克雷分布）和通用的数据聚类算法 PIC（幂迭代聚类）。后面两个算法是通过 MLlib 来调用的。我们也会介绍如何使用图来实现 MLlib 中基于矩阵的算法。我们发现使用图来表示输入数据，会增强标准机器学习算法的预测能力。最后，我们还会介绍一个用于半监督学习的定制算法，这在某种程度上是用图进行机器学习时最有趣的一个应用。

7.1 节介绍机器学习领域的必备知识：监督、无监督、半监督学习。接下来的章节介绍如何使用特定的算法来解决特定应用和场景带来的问题。深入地掌握机器学习方面的知识会让你在解决不同应用场景中的问题时更为得心应手，如选择合适的算法、改造和调节参数等，然而这部分内容不会包含在本书的范畴内。Peter Harrington 的 *Machine Learning in Action*（Manning，2012）是一本不错的入门书，Kim Falk 的 *Practical Recommender Systems*（Manning，2016）一书详细介绍了实用的推荐算法，提供了比本书 7.2 节概述部分介绍得更为深入的细节和更为通用的处理方法。

什么是“人工智能”？

从 20 世纪 50 年代开始，人们一直希望可以创造出类似人类智慧和能力的计算机。数十年来一些宣称成功的夸大言论，使得人们对这个概念感到疲劳，也造成了几次研究经费捉襟见肘和科研兴趣萎靡的“AI 寒冬”。早期人工智能的尝试包括搜索状态空间，实现启发、统计学、符号学逻辑，这些尝试取得了有限的成果。机器学习则关注最优化问题。在这个相比“创造出类似人类智慧和能力”更为现实的目标下，机器学习取得了巨大的成功，并在大数据时代得到广泛的应用。由于人工智能的概念已被打下不好的烙印，现在机器学习算法都避免提及与它之间的关系。

即使是那些尝试在机器上实现仿造人类智慧的人现在都会避免使用人工智能这个概念。相反，他们更偏向于使用强人工智能（AGI）和超人工智能（ASI）的说法。似乎现在人工智能更多的是指 20 世纪 70 年代时的一些方法论，但从学术定义上来讲，机器学习仍然是人工智能的一个子领域。

7.1 监督、无监督、半监督学习

在本节，我们将机器学习分成三个类目并在接下来的章节中介绍各类目的一些

例子。初看上去，机器学习可以分成两个大类目：监督学习和无监督学习。之后，一部分人发现可以使用半监督学习来结合两者的优势。

图 7.1 阐明了监督学习和无监督学习的差异。监督学习会给我们提供需要进行预测的不同物品的数据，例如一张图片描述的是一只猫还是一只狗。我们把这些数

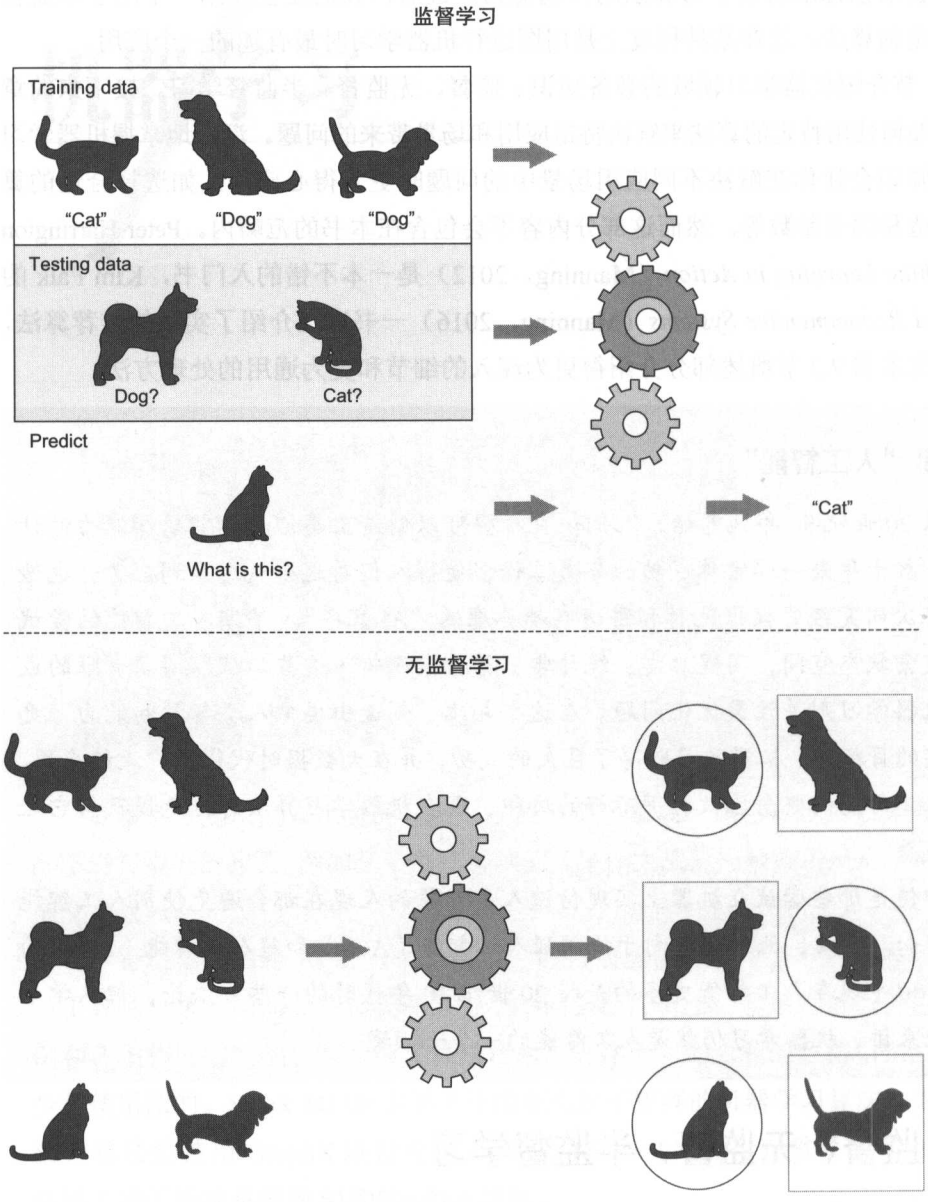


图 7.1 无监督学习将相似的数据分在同一类，但不能确定它们的标签。

据称为标签数据，因为对于每张图片都有一个标签——猫或者狗。我们可以使用这些数据去训练一个算法来预测未知图片的标签。

相反，无监督学习在我们不知道数据包含了什么内容时使用。它将相似的内容聚集在同一群组里，但我们并不一定知道每个群组代表的意义。无监督学习一般用于这种类型的聚类应用，但是监督学习适用于多种不一样的应用场景，包括分类、时间序列预测和推荐系统等。

对于监督学习和无监督学习，它们的目标都是训练机器学习的模型。一般我们拥有了一个训练好的模型，就可以使用它对新数据进行预测。在图 7.1 所示的使用监督学习模型时，事先未知标签的猫图片会被预测为“猫”。这个模型并不一定总是正确的。而无监督学习模型，会使用圆圈（也可以使用被算法自动标注为“组 1”的标签）或者正方形（也可以使用标注为“组 2”的标签）来作为预测输出，但由于模型是基于无标签数据训练出来的，它无法给出人类可理解的类似“猫”这样的标签。

无监督学习的优势是，无标签数据的获得更为容易和廉价。你可以使用自动的方式从网络上爬取这些数据。相比之下，标签数据则需要人工劳动。

本章涉及的算法和应用如下所述。

监督学习

- 基于 SVDPlusPlus 的影片推荐。
- 基于逻辑回归的网页垃圾信息检测。

无监督学习

- 基于 LDA 的话题聚类。
- 基于 K 近邻算法的图构建。
- 基于 PIC 的图像分割。

半监督学习

- 基于半监督学习的数据标注。

7.2 影片推荐：SVDPlusPlus

推荐系统是机器学习的一个热门应用场景。如果你想购买书籍或者影片，根据

历史购买记录你最有可能购买的是哪一个呢？或者是否可以根据你和其他消费者的相似度来计算呢？

本节将介绍唯一完全属于 GraphX 模块的机器学习算法 SVDPlusPlus 的用法（Spark 1.6）。和其他推荐系统算法一样，SVDPlusPlus 属于监督学习。

假设我们的任务是设计一个推荐系统进行影片推荐，我们拥有用户对已观看影片的历史评分，评分范围为 1 星到 5 星。可以使用图 7.2 所示的二分图进行说明，左边的顶点代表用户，右边的顶点代表影片，边则表示评分情况。虚线的边代表了需要进行的预测：Pat 会给《傲慢与偏见》打多少分呢？

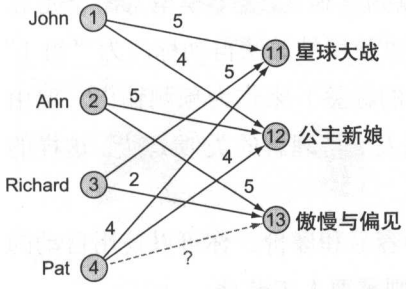


图 7.2 影片推荐。预计 Pat 对《傲慢与偏见》的打分情况如何呢？（边上的标签代表了 1 星到 5 星的评分，顶点上的数字代表了顶点 ID，稍后我们会用这些 ID 来替代文本）。

这个问题的另一种表示方式是使用邻接矩阵。一般来说，推荐系统的数据会构成一个*稀疏矩阵*，如图 7.3 所示，矩阵中大部分的项为空。SVDPlusPlus 会在内部将图转换成稀疏矩阵的表示方式。很多 SVDPlusPlus 相关的术语都会提到矩阵表示，而不是图表示。

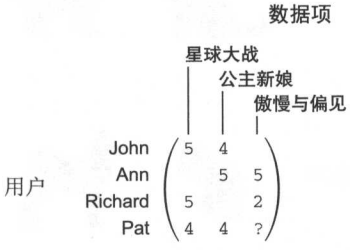


图 7.3 使用稀疏矩阵表示图 7.2 中的图。由于不是所有位置都有数字，所以这个矩阵被称为稀疏矩阵。在我们的小示例中，只有 4 个位置缺失数字（包括带问号待求解的项），但在一个典型的大型样本中，例如 100 万个用户和 10 万部影片，绝大多数位置上的数字都是缺失的。包括 SVDPlusPlus 在内的推荐系统，经常会使用矩阵来表示图。

推荐系统就是监督学习的一个例子，因为它提供了一些影片评分的数据，并要

求预测未知的用户对影片的评分。一般有两种主流的方法来解决这个问题。

第一种主流方法比较直接和简单：对于需要处理的用户，Pat，找到和他有相同爱好的其他用户，然后给 Pat 推荐这些用户喜欢的影片。这就是 Netflix 公司早期的推荐策略，有时被称为*邻居法*（neighborhood approach），因为它使用了图中相邻用户的信息。这种方法的一个缺点是，有时我们难以找到一个合适的邻居，就跟 Pat 的情况一样。同时，这种方法也忽略了影片自身的一些潜藏信息，而我们通常可以从一些可能并不相似的用户身上收集得到这些信息。

第二种主流方法是去挖掘一些隐性变量（latent variables），避免了第一种方法需要找到与目标用户准确匹配的其他用户的要求。这听起来会有点晦涩难懂，但它的基本原理并不复杂，如图 7.4 所示。通过隐性变量，我们就可以使用一个向量来表示每一部影片，向量代表电影拥有的不同特性。我们的例子使用了两个隐性变量，这样每部电影就可以使用一个二维向量来表示。尽管从图 7.4 看来，《星球大战》只具有科幻电影这一个特性，但是我们依然用一个长度为 2 的向量来表示它，第一个维度表示它属于科幻电影的程度，第二个维度表示它属于浪漫电影的程度。我们可以预期的是，第一个维度的值会相当高，第二个维度的值则会相当低，然而一般也不会是 0。

我们使用术语隐性（latent）来修饰这些变量，是因为这些变量并不是直接包含在输入的评分数据中。算法会根据用户喜好去“推断”出影片的通用特性。

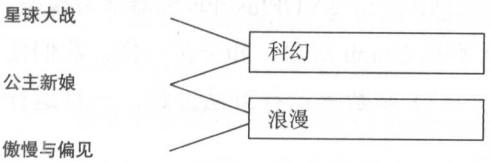


图 7.4 尽管在我们的例子中输入数据并不具有影片的风格信息，隐性变量可以自动推断出影片的风格，或其他相近的信息。算法无法给出确切的如“浪漫”这样的标签，但可以推断出《公主新娘》和《傲慢与偏见》在某些方面具有相似性。我们在图中给出了一些人类可理解的标签，但实际上隐性变量不会具有这些由用户添加或用其他类似方法得到的标签。事实上，算法生成隐性变量的过程在一定程度上并不可以解释，这些隐性变量也可能并不是我们所理解的“风格”。它有可能代表了“哈里森·福特电影”类型，或者稍微有点奇怪的“朋克喜剧”风格，或者是“粗鄙语言”类型等。

第二种方法使用了自动挖掘出来的隐性变量，因此它考虑到了全局的信息。即使是那些与 Pat 并不相似的用户，他们的喜好也会对每部电影的隐性变量造成影响，在为 Pat 做推荐时这些信息会被间接利用上。当然，这种方法也有一个缺点：它并

不像第一种简单的方法一样充分考虑到用户自身的信息。例如，如果 Pat 和他最好的朋友的喜好完全一致，那我们应该把这个最好的朋友看过而 Pat 没看过的电影推荐给 Pat。第一种基于寻找相似用户的方法可以很好地完成这个任务，但第二种基于隐性变量的方法则不一定能做得很好。

SVD++ 算法也是基于隐性变量的，但比上面的方法有所改进，不单单关注评价的分数，还找到一种方法利用隐含的信息。隐含信息体现在用户是否会对一部电影进行评分，尽管分数有可能很低，它也表达了电影的一些特性。例如，相比其他的科幻电影，有一个用户对《魅影危机》打了很低的分数。但是，《魅影危机》也被评分了的事实，说明了它和这个用户评分过的其他电影具有某些方面的联系。

2008 年，Yehuda Koren 在 论 文 *Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model* 中提出了 SVD++ 算法，在 Spark GraphX SVDPlusPlus 的文档中有相关链接。由于 Spark 文档关于 SVDPlusPlus 的介绍相对比较少，建议读者自行阅读上述论文，里面介绍了推荐算法的背景知识，包含了相关的定义、概念和公式。除了介绍 SVD++，论文还介绍了一种结合相似用户和商品信息提升推荐效果的扩展方法，同时考虑了三方面的因素：隐性变量、隐含信息和相似用户（再重申一次，标准的 SVD++ 算法是基于隐性变量和隐含信息的）。我们对 Koren 的论文进行了较多的说明，但如果你想更深入地了解细节，欢迎对原文进行更仔细的研究。

清单 7.1 展示了如何使用图 7.2 所示的图去训练一个 SVDPlusPlus 机器学习模型。算法的输入是一个代表了图的 EdgeRDD，而不是 Graph 对象。和平常一样，我们使用算法对象（本例中为 `SVDPlusPlus`）的 `run()` 函数来运行算法过程。一旦运行完毕，我们就获得了两个可进行预测的模型。

清单 7.1 使用图 7.2 的数据来运行 SVDPlusPlus

```
import org.apache.spark.graphx._

val edges = sc.makeRDD(Array(
  Edge(1L,11L,5.0),Edge(1L,12L,4.0),Edge(2L,12L,5.0),
  Edge(2L,13L,5.0),Edge(3L,11L,5.0),Edge(3L,13L,2.0),
  Edge(4L,11L,4.0),Edge(4L,12L,4.0)))

val conf = new lib.SVDPlusPlus.Conf(2,10,0,5,0.007,0.007,0.005,0.015)

val (g,mean) = lib.SVDPlusPlus.run(edges, conf)
```

构建图7.2中的EdgeRDD。

算法的指定超参数（参考表7.1）。

运行SVD++算法，获得返回的模型——输入图的再处理结果和数据集的平均打分情况。

Scala 小贴士：虽然 Scala 不支持如 Python 那样的多个返回值方式，但是 Scala 提供了 `val` 声明语法来分解元组（例如函数返回的元组），并可以为元组内的成员赋予不同值（变量名）。

表 7.1 记录了 `Conf` 的参数，和建议的参数值。表中提及的 `gamma1` 和 `gamma6` 的偏差值，是针对 SVD++ 类型算法而设定的，在后续的章节中也会有进一步的说明。

表 7.1 参数设定

参数	示例	描述
`Rank`	2	隐性变量的个数
`maxIters`	10	执行的迭代数。值越大，模型会更有可能收敛到理想的解决方案，预测准确率也会越高
`minVal`	0	最低的评分（0 星）
`maxVal`	5	最高的评分（5 星）
`gamma1`	0.007	每次迭代中，偏差的改变速度，对应 Koren 论文里的 γ_1，推荐值为 0.007
`gamma2`	0.007	隐性变量的改变速度，对应 Koren 论文里的 γ_2，推荐值为 0.007
`gamma6`	0.005	偏差的阻尼系数，用于保持偏差值不会过大。对应 Koren 论文中的 λ_6，所以该参数命名为 lambda6 可能会更合适，推荐值为 0.005
`gamma7`	0.015	不同隐性变量被允许相互影响的程度。对应 Koren 论文中的 λ_7，所以该参数命名为 lambda7 可能会更合适，推荐值为 0.015

定义：Koren 论文中的 4 个参数 γ_1、γ_2、λ_6 和 λ_7（对应 GraphX `SVDPlusPlus.Conf` 中的 `gamma1`、`gamma2`、`gamma6` 和 `gamma7`），是机器学习中会用到的超参数（hyperparameter）。有人认为这是“容差系数（fudge factors）”的另一种奇特说法。在训练开始前，超参数就会被确定下来。调参过程主要是以经验为主。预先知道如何设定这些超参数是一件困难的事情，所以你需要参考在其他应用中使用过这个算法的经验，或者在自己的应用中不断进行尝试。

在这个例子中，基于给定的三部影片有两种不同的流派类型，我们将秩设为 2。这意味着，对于每部影片，算法的隐性变量的长度也相应为 2，第一维（有可能）表示它的科幻属性，第二维则（有可能）代表浪漫属性。更大的数据集里可能会有更多的影片，这时秩的典型取值可以是 10、20 或者是 100。

以下是输入的参数。清单 7.2 展示了如何利用 SVDPlusPlus 产生的模型进行预测。它定义了 pred() 函数，输入为模型的参数、用户 id 和我们希望进行预测的影片。在这个例子里，我们使用这个函数来预测 Pat 对《傲慢与偏见》的评分，获得的结果是 3.95 星。

清单 7.2 函数 pred() 以及调用方法

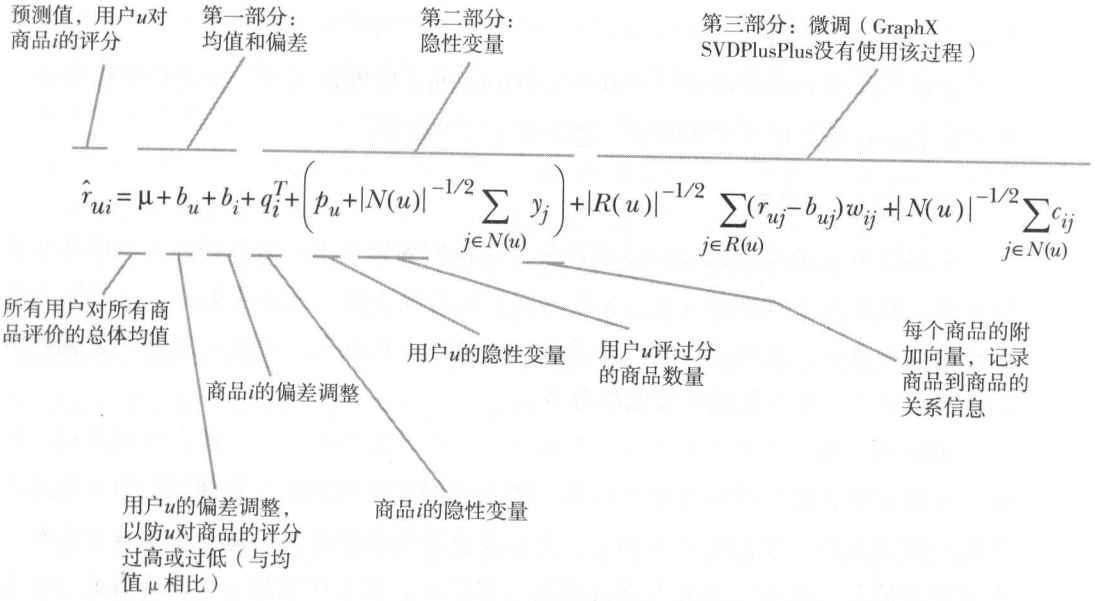
```
def pred(g:Graph[(Array[Double], Array[Double], Double, Double),Double],
         mean:Double, u:Long, i:Long) = {
  val user = g.vertices.filter(_._1 == u).collect()(0)._2
  val item = g.vertices.filter(_._1 == i).collect()(0)._2
  mean + user._3 + item._3 +
    item._1.zip(user._2).map(x => x._1 * x._2).reduce(_ + _)
}
pred(g, mean, 4L, 13L)
```

Scala 小贴士：使用 Scala 时，可通过 zip()、map() 和 reduce() 这几个函数来计算点积。

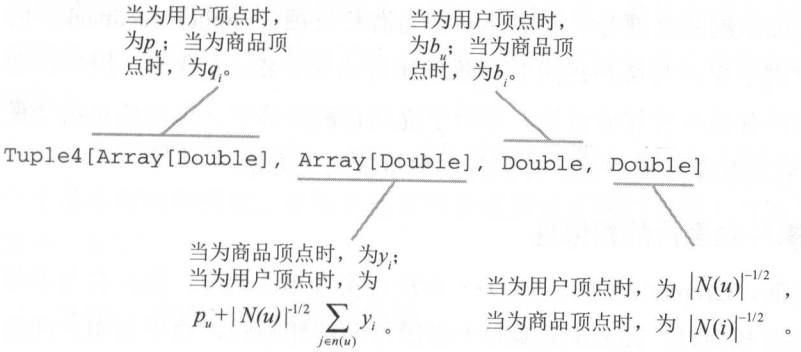
注意：SVDPlusPlus 的一部分初始化是使用随机数的，所以对于同一份输入数据，每次程序的预测结果不一定完全一样。

你可以简单地参考清单 7.2 中的实现来使用 pred() 函数，但如果希望更深入地知道模型是如何生效的，则需要理解它的构建过程。首先，我们给出了论文里的 SVD++ 的相关公式。然后，我们对照 GraphX SVDPlusPlus 的返回结果来理解它们的关系。最后，我们将论文中公式里的变量和 SVDPlusPlus 的返回值进行关联，来构建 pred() 函数。

在 Koren 的论文里，预测公式如下：



进行预测时，我们需要使用调用 SVDPlusPlus 后返回的模型。正如清单 7.1 所示，SVDPlusPlus 会返回一个包含两个值的二元组：一个图模型，一个代表全图平均评分的双精度数值（对应上述公式里的 μ）。返回的新图的边属性值和原始图的一致，代表已知的用户评分。然而新图的顶点集，则是比较复杂的四元组：



在介绍过论文中的公式和 GraphX SVDPlusPlus 的返回内容后，我们来研究该如何构建 `pred()` 函数和计算用户 *u* 对影片 *i* 的评分。函数 `pred()` 的返回值对应论文公式的前 4 个元素。第 5 个元素被省去了，因为 GraphX SVDPlusplus 没有实现算法的“第 3 部分”，即考虑邻居影响的微调。

7.2.1 公式解释

使用 `pred()` 即可构建一个基于 SVDPlusPlus 的推荐系统。但是如果你想进一步了解 Koren 论文里公式的原理，建议阅读以下内容。

第 1 部分：偏差

全局均值 μ 是基于已知的所有评分。在我们的例子中，影片的评分范围从 0 星到 5 星，则理论上预期的 μ 是 2.5 星，但实际情况下很可能并不如此。这是因为当用户进行评分时，很可能会有一个总体向上或向下的偏差，要么总体用户会偏向给出较高的评分，要么是偏向较低的分数。

相似的，每一个用户也会有一个偏差。一个特定的用户有可能会比较挑剔，则他有可能对所有影片的评分都比较低。那么在对这个用户进行处理时，得到他对各个影片的评分后，我们还会再加上一个偏差值得到最终结果。每一个影片也会有一个相应的偏差。如果大部分人都讨厌某一部影片，那么在预测一个尚未看过这部电影的用户的评分时，我们需要将预测评分值适当降低。这是由偏差变量 b_i 实现的。

第 2 部分：隐性变量

同样的，隐性变量的长度是由 `Conf` 里的秩参数（`Rank`）决定的。在预测公式里，所有的隐性变量用 p_u 来表示，它的长度就是秩参数。

第 2 部分里被标示的其余部分，和 Koren 称为潜在反馈（implicit feedback）的内容相关，在这个例子里，与用户已进行评价的影片有关。Koren 认为，用户已评分的影片携带了除评分值外的其他信息，可用于提高预测准确率。这些信息由变量 y_j 代表，在公式里它的权重是用户已评分影片的数量值的平方根。

第 3 部分：影片和影片的相似性

正如前面提到的，GraphX 版本的 SVD++ 并没有实现第 3 部分，即没有考虑邻居相似性。对于邻居相似性，Koren 更偏向于指影片间的相似性，而不是用户间的相似性。第三部分公式的前半部分 w_{ij} 代表了影片 i 和影片 j 的相似度，后半部分则着重考虑潜在的反馈，在一定程度上和第二部分的潜在反馈所指内容类似。

7.3 在MLlib中使用GraphX

Spark MLlib 中包含了许多机器学习算法，本节会介绍其中使用了 GraphX 的两

个算法，并会介绍如何结合图进行基于矩阵的 MLlib 监督学习。

SVDPlusplus 是唯一一个完全被包含在 GraphX 模块中的机器学习算法，但还有其他的两个算法：隐含狄利克雷分布（LDA）和幂迭代聚类（PIC），也是基于 GraphX 进行构建的。但最终它们被归属到 MLlib 模块中，而不属于 GraphX。我们首先介绍如何使用无监督学习方法 LDA 来确定文档的主题，然后介绍如何使用同样属于无监督学习范畴的 PIC 来进行图像分割，它用于计算视觉领域。

而对于另一个应用——检测网页垃圾信息，我们将介绍另一个 MLlib 算法——LogisticRegressionWithSGD，它一般情况下和图并无直接联系，但可以和 GraphX 的 PageRank 算法配合使用，进行网页垃圾检测。

7.3.1 主题聚类：隐含狄利克雷分布

假设你有大量的文档并希望确定每篇文档的主题，这时候 LDA 就派上用场了。在接下来的章节中我们会计算 20 世纪 80 年代路透社电报新闻的主题。LDA 属于无监督学习，所有的主题并不需要事先指定，是在聚类过程中逐渐形成的。

MLlib 的 LDA 使用了 GraphX 来提高计算效率，尽管它的输入和输出都不是图。正如它的名字所示，LDA 是基于隐含变量的，在这里隐含变量指的是算法自动推断出来的“主题”。这些主题由一些与之关联的单词描述，但并不具有明确的主题名称。一般情况下，需要人工检验每个主题包含的单词，然后为每个主题指定一个有意义的名称。正如图 7.5 中的例子所示，这些有意义的名称会被附加在每一个主题单词清单上。

一旦主题被推断出来，每篇文档在不同主题上会有不同的分数。这是 LDA 的一个基本原则和假设：每篇文档同时会在所有不同的主题上有所涉及，而不仅仅是其中一两个。

示例：路透社电报新闻分类

LDA 以一系列的文章作为输入，输出时则提供了主题的清单（每一个主题会有关联的单词清单），以及每篇文章与每个主题的关联强弱程度。

在这个例子中，我们下载了路透社 1987 年的电报新闻，使用 LDA 来推断它们的主题清单，以及计算每篇新闻与不同的主题的关联程度分数。清单 7.3 展示了如何从加利福尼亚大学下载数据，并展示了如何使用 Linux/OS X 的命令对数据进行清洗。在进行数据清洗时使用了 sed 和 tr 这样的便捷工具。如果你对这些命令不熟悉，

可以查找相关的书籍或网络资料来学习。路透社的数据采用的是SGML格式，这是HTML格式的前身。对于每一篇新闻，我们关注的是<BODY>和</BODY>之间的文本内容。为了方便Spark进行文本文件的处理，每一行对应一篇新闻（所以有些行的内容长度会比较长）。

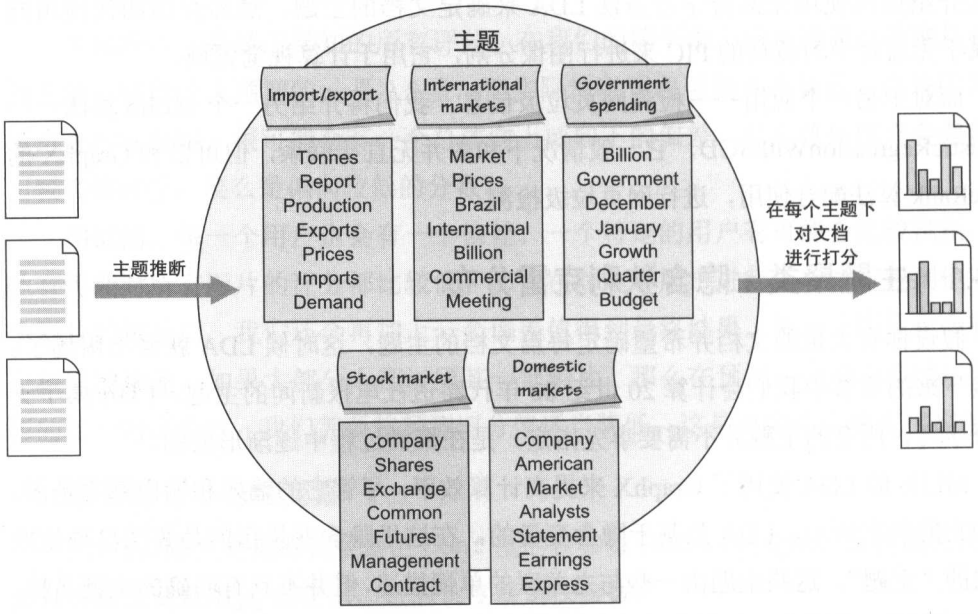


图7.5 隐含狄利克雷分布。隐含变量指的是各个主题，并由算法自动生成。主题名称是经过人工推断附加的，算法本身并没有为主题赋予名称的能力。每一个文档在不同的隐含变量（主题）上的程度有所不同。

清单7.3 路透社电报新闻的下载与清洗

```
wget https://archive.ics.uci.edu/ml/machine-learning-databases/
➥ reuters21578-mld/reuters21578.tar.gz
tar -xzvf reuters21578.tar.gz reut2-000.sgm
cat reut2-000.sgm | tr '\n' ' ' | sed -e 's/<\/BODY>/\n/g' |
➥ sed -e 's/^.*<BODY>//' | tr -cd '[[:alpha:]] \n' >rcorpus
```

尽管上述方法完成了数据清洗的大部分工作，但数据的预处理还需要进一步的操作。Spark的LDA需要将文档表示为词袋（bag of words）的形式，这是一种通用的机器学习表达形式，如图7.6所示。当进行词袋生成时，我们首先会过滤掉停用词（stop words），它们在文档中的存在过于普遍因而携带的信息量极为有限。

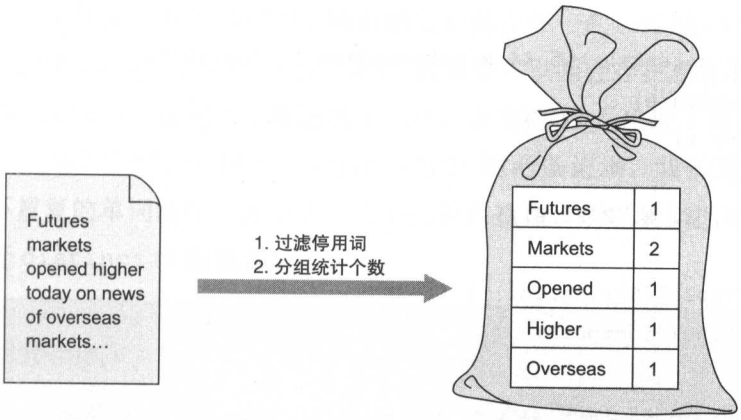


图 7.6 文档的词袋表示形式。

拥有一个适合的停用词词表对 LDA 而言相当重要，可以避免被一些无关的内容影响。然而，清单 7.4 里的函数 `bagsFromDocumentPerLine()`，我们没使用停用词词表，而是简单地将一些较短（少于或等于 5 个字母）的单词过滤掉，同时也会过滤掉“Reuter”这个特殊单词。

请注意，在函数 `bagsFromDocumentPerLine()` 里，所有内容经过 `split()` 操作后会转化成传统的 Scala 集合（而不是 RDD）。因此，后续采用的是 `groupBy()` 这种 Scala 集合上的操作方法，而不是 RDD 上的操作方法 `groupByKey()`，尽管它们功能相似。

按照 Spark 的 LDA 所期待的输入形式，我们不会提供原始的字符串，而是提供在全局词典里各单词的索引。我们首先会进行全局词典的构建，对应接下来清单中的 `vocab` 变量。我们会把它作为 driver 的一个本地变量，而不是以 RDD 的形式存储。因此，你会注意到词典具有 `flatMap()` 这种形式的操作，这是一个通用的函数式编程方法，在下面会有详细说明。

清单 7.4 在路透社电报新闻上运行 LDA 算法

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.clustering._
import org.apache.spark.rdd._

def bagsFromDocumentPerLine(filename:String) =
  sc.textFile(filename)
```

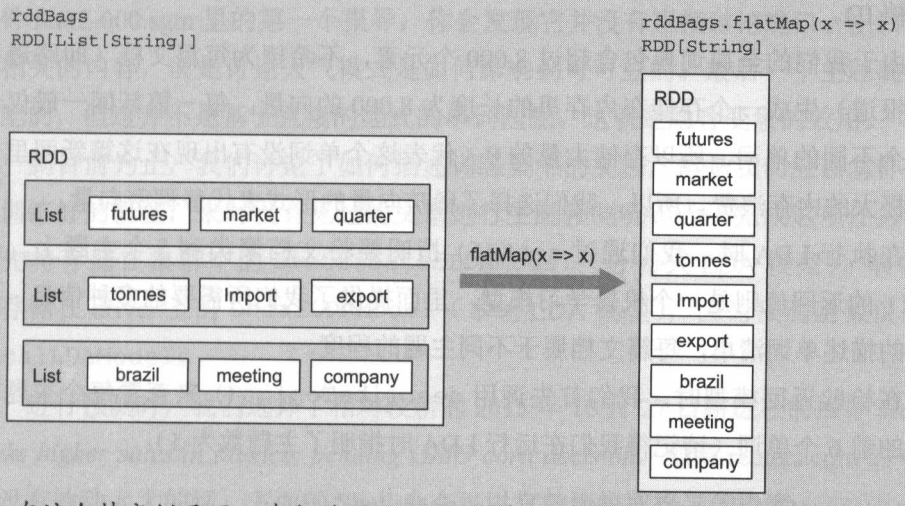
```
    .map(_.split(" ")
          .filter(x => x.length > 5 && x.toLowerCase != "reuter")
          .map(_.toLowerCase)
          .groupBy(x => x)
          .toList
          .map(x => (x._1, x._2.size)))

val rddBags:RDD[List[Tuple2[String,Int]]] =
  bagsFromDocumentPerLine("rcorpus")

val vocab:Array[Tuple2[String,Long]] =
  rddBags.flatMap(x => x)
         .reduceByKey(_ + _)
         .map(_._1)
         .zipWithIndex
         .collect

def codeBags(rddBags:RDD[List[Tuple2[String,Int]]]) =
  rddBags.map(x => (x ++ vocab).groupBy(_._1)
                               .filter(_._2.size > 1)
                               .map(x => (x._2(1)._2.asInstanceOf[Long]
                                                    .toInt,
                                          x._2(0)._2.asInstanceOf[Int]
                                                    .toDouble))
                               .toList)
         .zipWithIndex.map(x => (x._2, new SparseVector(
                                              vocab.size,
                                              x._1.map(_._1).toArray,
                                              x._1.map(_._2).toArray)
                                         .asInstanceOf[Vector]))
val model = new LDA().setK(5).run(codeBags(rddBags))
```

Spark 小贴士：Spark 在 org.apache.spark.mllib.linalg 中定义了 Vector 的数据结构，但它和 Java 的 java.util 中的 Vector 并不一样。正如名字所示，Spark 的 Vector 是属于 MLlib 的特定结构。它是抽象类型的，并且有两种具体的实现形式：DenseVector 和 SparseVector（稠密向量和稀疏向量）。

在词典计算的过程中，所有的文档混合在一起，经过 flatmap 的操作后，会得到一个 Tuple 2[String, int] 的集合，里面会存在重复的元素。举例说明，单词 commerce 在文档 5 里出现了三次，在文档 12 里出现了两次，则在 RDD 里会有 ("commerce",3) 和 ("commerce",2) 这两条记录。我们最终期望得到的是一个不重复的单词清单，并不关心它们出现的次数，所以我们把次数的信息省去，并执行 distinct() 的操作。

flatMap()

尽管函数名 flatMap() 是由 Scala 提出的，但由于它解决的是一个通用问题，在其他的函数式编程语言里也会有相对应的函数。map() 函数将输入的每一个元素一一对应转化成输出的另一个元素，flatMap() 完成的则是将输入的每一个元素转换成输出的 0 个、1 个或者多个元素组成的序列。每一个序列然后会被“打平”，组合成为一个序列。在大部分情况下，flatMap() 的效果与 map() 操作后再进行 flatten() 操作的效果相近（在包含不同数据类型的情况下不成立）。Spark 提供了针对 RDD 的 flatMap() 操作，在构建词典时的作用如以下插图所示。



在这个特定例子里，我们使用 x => x 作为 flatMap() 的转化方法，这使得我们的操作与 flatten() 更为相似。但是尽管 Scala 拥有 flatten() 方法，Spark 没有针对 RDD 的对应操作，所以我们仍然使用 flatMap(x=>x) 来实现需求。

flatMap() 对单维的数据也是极为有用的，因为传递给这个函数的参数有机

会被转换成为一个个独立的集合。例如，一种可行的构建词典的方式是 sc.textFile("rcorpus").flatMap(_.split(" "))，这会一下子将所有文档转化成为多个独立的单词，并将它们组成一个统一的单词清单（代表所有文档合在一起的结果）。

当词典构建完成后，我们就可以将数据按照 Spark LDA 所需要的形式组织起来了。codeBags() 函数完成了两件事情：前半部分是将词典里字符串形式的单词转换成对应的索引下标，后半部分是将它转换为 LDA 可处理的稀疏向量（SparseVector）形式。

在转换词典索引下标时，codeBags() 前半部分进行的操作，概念上和 SQL INNER JOIN 或者 Spark RDD 的 join() 操作是一致的。但是 rddBags 里的每一个元素都是一个 Scala 的 List 对象，不可进行 join() 处理。由于 Scala 里缺少 join() 的处理方法，我们的处理方式是，基于 groupBy() 函数，使用 ++ 操作进行两个数组的合并，再将没有同时出现在两个清单中的元素去掉。

为了完全满足 Spark LDA 的输入形式，codeBags() 的后半部分操作又完成了两件事：首先将词袋转换成稀疏向量，然后使用 zipWithIndex() 给每篇文档附上文档 ID。

由于我们的全局词典包含超过 8,000 个元素，不希望为每篇文档（即路透社的新闻报道）生成一个存储在内存里的长度为 8,000 的向量。每一篇新闻一般仅包含数十个不同的单词，所以存储大量的 0（代表这个单词没有出现在这篇新闻里）会造成极大的内存浪费。所以，我们选择了稀疏向量的形式来代替稠密向量。

在执行 LDA 时，我们通过 setK(5) 指明要将文档聚类到 5 个主题上。函数 run() 的返回值则是一个机器学习模型，里面提供了我们所需要的多种信息：每个主题的描述单词清单，每篇文档属于不同主题的程度。

在检验返回模型时，我们首先调用 describeTopics() 来查看每个主题的最相关的前 6 个单词（请记得我们在运行 LDA 时指明了主题数为 5）。

```
model.describeTopics(6).map(_._1.map(vocab(_)._1))
res1: Array[Array[String]] = Array(
  Array(profit, company, billion, president, products, treasury),
  Array(market, japanese, billion, international, brazil, american),
  Array(billion, january, december, government, quarter, growth),
```

```
  Array(company, shares, exchange, common, interest, securities),
  Array(tonnes, prices, production, billion, exports, system))
```

现在，让我们来看看第一篇文档的主题分布情况。由于 topicDistributions() 函数会改变文档的顺序，所以我们需要用 filter() 来获得文档 ID 为 0 的文章：

```
model.asInstanceOf[DistributedLDAModel].topicDistributions
     .filter(_._1 == 0).collect
res2: Array[(Long, org.apache.spark.mllib.linalg.Vector)] =
    Array((0,[0.050547152608283755,0.05217337794656473,0.041732176735789286,
0.0418304122726957,0.8137168804366666]))
```

Spark 小贴士：由于 Spark 采用 REPL（交互式编程环境）的机制，进行 DistributedLDAModel 的类型转换是必需的。如果是代码编译的方式，则不需要进行强制转换。

对于第一篇文档，它与最后一个主题的关联程度最为强烈——分值为 0.81，而其他 4 个主题的分值都没有超过 0.10。第一个主题所代表的含义已经在之前通过 describeTopics() 的方式查看过：进口 / 出口和商品报导。有趣的是，查看原始文件 reut2-000.sgm 里的第一个报导，你会发现它并没有直接讨论进口、出口或者商品相关的内容，而是讨论天气模式是如何影响到可可豆的。最后一个主题确实是最匹配的，但这并不是基于直接的显式的单词匹配。这就是隐含变量的效用。

到目前为止，我们讨论了如何描述训练集中的文档，但是我们应该怎样使用这个训练好的模型，来对没有见过的文章进行主题预测呢？为了完成这项任务，我们首先将存储在集群中的 DistributedLDAModel（分布式 LDA 模型），转换成完全存储在 driver 上的 LocalLDAModel（本地 LDA 模型），因为预测函数仅可使用 LocalLDAModel。

进行预测时，我们选择了相对较新的 2015 年 10 月 27 日路透社的报导 *Monsanto seeks higher sales in Mexico, pending GMO corn decision*。如果 reuters.com 的 CSS 样式没有改动太大的话，下面的 Shell 命令可以有效地提取报导的内容：

```
wget http://www.reuters.com/article/2015/10/28/
➥ mexico-monsanto-idUSL1N12S00D20151028 -O Mexico
cat Mexico | tr '\n' ' ' |
➥ sed -e 's/^.*midArticle_start">//;s/<script.*$//;s/<[^>]*>//g' |
```

```
➥ tr -cd '[[:alpha:]] \n' >Mexico2
```

然后将模型转换为 LocalLDAModel 并进行主题预测：

```
model.asInstanceOf[DistributedLDAModel].toLocal.topicDistributions(codeBags(b
    agsFromDocumentPerLine("Mexico2"))).collect
res3: Array[(Long, org.apache.spark.mllib.linalg.Vector)] = Array(
(0,[0.20745026733300995,0.1957256951092684,0.1664498204599232,
0.17574123052282312,0.25463298657497535]))
```

所有的取值都在 0.2 附近，说明我们基于 20 世纪 80 年代的报导训练的模型，在预测 2015 年的路透社报导时效果并不是那么好。很明显，词典和主题在 30 年的时间内发生了巨大的变化。但是最后的主题的分值是 0.25，这个主题讲述的是进口 / 出口相关的内容，而这也确实是待预测文章所描述的。

如何在 LDA 实现中使用图

正如上面使用 LDA 的过程所描述的，它并没有使用到图进行处理。如果你想知道如何使用图来提高算法性能，下面的内容会介绍基于图来进行 Spark LDA 的实现。这个介绍会着眼在较高的层次，而不深究细节。希望进行更深入的理解时，请参考 Asuncion 等人在 2009 年发表的文章 *On Smoothing and Inference for Topic Models*。

在 Spark 1.6 中，LDA 可以使用两个不同的算法来实现：默认的为基于图的最大期望（EM）算法，或者是在线变分贝叶斯算法。在这里我们会着重讨论 EM 算法。EM 算法被用于解决存在有未知或隐藏变量的概率问题，正如 LDA 模型里的隐含变量。LDA 建立了一个概率方程组，如一个单词属于一个特定主题的概率和一个单词在一篇特定文档中的概率。

求解上述概率方程组时，EM 算法的两个主要步骤是，期望和最大化，运作的方式与 K-Means 方法类似。在期望步骤中，我们会对一些变量进行猜测或估计。然后，在最大化步骤中，计算对应的误差，这些变量会被相应地进行调整。这样的过程经过多次迭代后，在期望步骤中做出的猜测会越来越接近真实的概率值。

这与图有什么关联呢？

尽管 EM 算法解决了概率方程组问题的期望最大化算法，但它本身并不是一个图算法，Spark LDA 在实现过程中使用了 GraphX 来提高计算性能。这和 7.1 节讨论的 Spark SVD++ 实现很相似，使用图来代替稀疏矩阵。使用了图的 EM 算法有效地

提高了 LDA 的性能，因为任何一篇给定的文档都只用到了全局语料词典的一个小型子集。这种思路如图 7.7 所示。于是，在计算总和时，只有相关联的项被求解了。而诸如“乘以零再求和”（multiplyby-zero-and-add）等在非图实现时会采用的步骤，会被完全跳过。

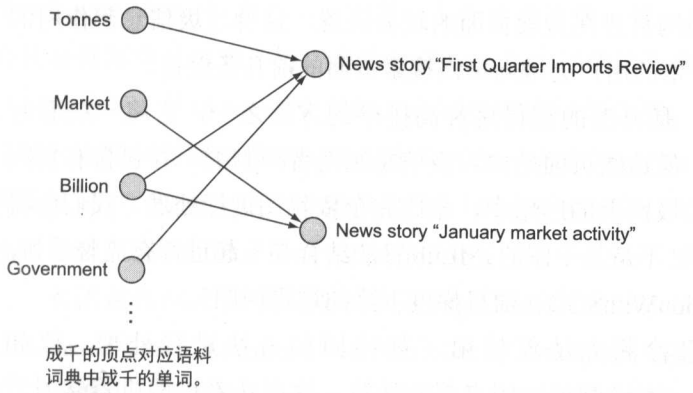


图 7.7 不是语料词典中的所有单词都会出现在每一篇文章中。图表示的方法可以提高计算性能。

LDA 参数

表 7.2 罗列了 Spark LDA 的 4 个主要参数，在构建 `LDA()` 对象后有这些参数对应的 getters 和 setters 方法。

表 7.2 LDA 参数

名字	默认值	描述
`k`	`10`	主题数目
`maxIterations`	`20`	迭代次数（这里，numIterations 可能会是一个更合适的命名）
`alpha`	`50/k + 1`	文档集中度，使用 EM 算法时取值要求大于 1.0。取值越高，则文档会与越多的主题相关，甚至与全部主题相关
`beta`	`1.1`	主题集中度，使用 EM 算法时取值必须要大于 1.0。取值越高，则主题会与越多的单词相关，甚至与全部单词相关。相反，当取值较低时，每个主题只会包含少数单词

上述推荐的 alpha 和 beta 的取值，借鉴了 Griffiths 和 Steyvers 在 2004 年发表的论文 *Finding Scientific Topics*，并参考了 Asuncion 在 2009 年发表的论文，进行了额外的 +1 调整。

7.3.2 垃圾信息检测：LogisticRegressionWithSGD

相信大部分读者对垃圾邮件都不陌生，但事实上存在有多种不同类型的垃圾信息。在本节，我们集中考虑网页垃圾——内容上和垃圾邮件相似的网页，垃圾信息制造者使用了不同的技术来引诱我们进入这些网页。一般而言，他们会操作搜索引擎的排序结果，使得他们的网页排在搜索页面的较前位置。这种垃圾信息起作用的原因在于，一般使用搜索引擎的用户会偏向单击搜索页面的前几条链接。

链接工厂（link farms）是典型的获得这种高排序的方法之一，它将一些看起来正常的页面聚集在一起，但这些页面是由垃圾信息制造者控制的，并包含有指向垃圾网页的链接。在检测垃圾网页的应用中，相比完全依赖图进行处理，我们会将PageRank算法和一个通用的(不是基于图的)MLlib算法结合在一起进行有监督学习，这个算法是LogisticRegressionWithSGD（随机梯度下降的逻辑回归）。

很多简单的垃圾信息检测方法仅使用了某种回归方法进行处理，比如LogisticRegressionWithSGD，而没有使用图来帮助计算。这意味着这些机器学习方法仅将网页看作独立的个体，而忽略了它们彼此间的链接关系。这些不基于图的方法无法解决如图7.8所示的链接工厂问题。

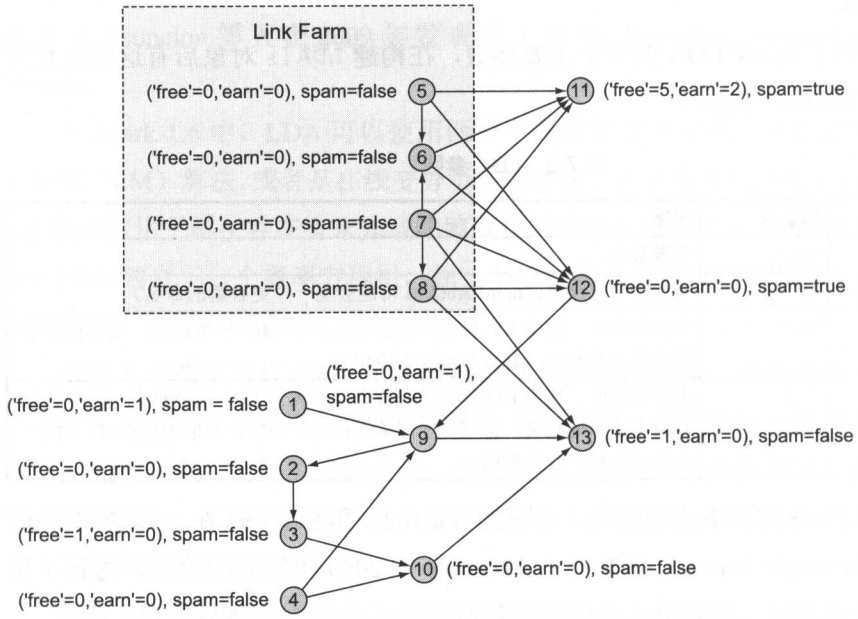


图7.8 网页垃圾训练数据。每一个顶点代表了一个网页，附上单词free和earn出现的次数，和人工打标的是否属于垃圾网页的标签。在垃圾词汇数据基础上再加上PageRank的数据，我们就可以用链接工厂的模式来帮助确定网页垃圾。

逻辑回归和更为人们熟知的线性回归很相似。在线性回归中，它的输出，即预测值，是一个浮点数。而在逻辑回归中，预测值则是一个代表样本属于哪个标签的整数。我们的例子包含了两种可能的标签："垃圾信息"和"非垃圾信息"。在逻辑回归（和线性回归）中，每一个样本（即对应我们例子中的网页）都用一个特征向量（feature Vector）来表示。它是一个浮点数向量，每一个浮点数代表了一维特征。特征是机器学习中的术语，表示对样本的某些特性的衡量。在我们的例子中，使用了三种特征：单词 free（免费的）在网页中出现的次数、单词 earn（赚）在网页中出现的次数，和网站的*截断网页排名*（Truncated Page Rank）。在一个成熟的垃圾信息监测器中，会有上百个"触发单词"（trigger words），但在我们的简单的示例中仅使用了两个这样的触发单词：free 和 earn。

术语截断网页排名（Truncated Page Rank）由 Becchetti 等人在 2006 年的论文 *Link-Based Characterization and Detection of Web Spam* 中提出。它的作用是识别链接工厂的模式，通过在 PageRank 计算中去除直接相连的第一层顶点的影响，来实现这一修改的网页排名计算方法。Becchetti 的论文中描述了不同的 PageRank 迭代方式，对它们进行归一化、差异化，最终提出了截断网页排名算法。

在这里，我们采用一种更简单和有效的方法。我们运行 PageRank 算法两次，第一次仅进行一轮迭代，第二次进行五轮迭代，然后计算每个顶点的两个不同 PageRank 值的比例。这样做的目的是，我们期望与正常网页相比，一个垃圾网页会具有不同的网络拓扑结构。我们希望通过截断网页排名特征来捕获这一不同点。

通常而言，监督学习需要有训练集和测试集。我们的训练集如图 7.8 和清单 7.5 所示，测试集则请查看图 7.9 和清单 7.9。为了在 `LogisticRegressionWithSGD` 中使用图顶点数据，我们首先需要将它转化成算法所期待的输入格式，即存储着 `LabeledPoint` 格式数据的 RDD。`LabeledPoint` 格式是指特征向量再附加上人工打标的标签。请记住，在监督学习的训练过程中，我们会提供答案给机器学习算法进行学习。这些标签就是上一句提及的答案。最终期望的结果是在训练充分的情况下，当我们将一些新数据提供给这个训练好的机器学习模型时，它可以正确预测数据的标签，即当我们提供一个未经判断的新网页，算法可以给出它是否为垃圾网页的判断。

清单 7.5 根据图 7.8 构建训练集的图

```
import org.apache.spark.graphx._
```

```
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

val trainV = sc.makeRDD(Array((1L, (0,1,false)), (2L, (0,0,false)),
  (3L, (1,0,false)), (4L, (0,0,false)), (5L, (0,0,false)),
  (6L, (0,0,false)), (7L, (0,0,false)), (8L, (0,0,false)),
  (9L, (0,1,false)), (10L,(0,0,false)), (11L,(5,2,true)),
  (12L,(0,0,true)), (13L,(1,0,false))))

val trainE = sc.makeRDD(Array(Edge(1L,9L,""), Edge(2L,3L,""),
  Edge(3L,10L,""), Edge(4L,9L,""), Edge(4L,10L,""), Edge(5L,6L,""),
  Edge(5L,11L,""), Edge(5L,12L,""), Edge(6L,11L,""), Edge(6L,12L,""),
  Edge(7L,8L,""), Edge(7L,11L,""), Edge(7L,12L,""), Edge(7L,13L,""),
  Edge(8L,11L,""), Edge(8L,12L,""), Edge(8L,13L,""), Edge(9L,2L,""),
  Edge(9L,13L,""), Edge(10L,13L,""), Edge(12L,9L,"")))

val trainG = Graph(trainV, trainE)
```

在上述代码中，我们的图顶点仅含有三个特征中的两个（单词“free”和“earn”的出现次数），以及一个人工打标的标签（是否为垃圾网页的“true”与“false”）。它仍缺失第三个特征：截断网页排名。清单 7.6 计算了这一特征，并将它和其他顶点数据一起组装成 LogisticRegressionWithSGD 所需要的 LabeledPoint 形式。我们定义的 augment() 函数，支持了三个 RDD 的 join 操作——顶点数据、1 轮迭代的 PageRank 值、5 轮迭代的 PageRank 值，将它们组合在一个 LabeledPoint 里，并同时计算两个 PageRank 值的比例。LabeledPoint 格式会使用向量来存储特征数据，在这种情况下，我们偏向使用 DenseVector 而不是 SparseVector，因为我们的特征向量不会有缺失值，并且 DenseVector 的形式更方便处理。

清单 7.6 逻辑回归数据准备

```
import org.apache.spark.graphx.lib.PageRank
import org.apache.spark.mllib.linalg.DenseVector
import org.apache.spark.mllib.regression.LabeledPoint

def augment(g:Graph[Tuple3[Int,Int,Boolean],String]) =
  g.vertices.join(
    PageRank.run(trainG, 1).vertices.join(
      PageRank.run(trainG, 5).vertices
```

```
    ).map(x => (x._1,x._2._2/x._2._1))
  ).map(x => LabeledPoint(
    if (x._2._1._3) 1 else 0,
    new DenseVector(Array(x._2._1._1, x._2._1._2, x._2._2))))
```

注意：尽管 LabeledPoint.label 被定义为 Double 类型，逻辑回归要求的输入是从 0 开始的整型数值：0.0、1.0、2.0 等。这是由于在 Spark 线性回归中也会使用到 LabeledPoint，所以它被定义为 Double 类型。

现在我们开始训练逻辑回归模型，如清单 7.7 所示。在这里，我们仅将 LogisticRegressionWithSGD 执行 10 轮迭代作为示例。一般而言，SGD（随机梯度下降）有时需要上百轮迭代才可以收敛。

清单 7.7 训练逻辑回归模型

```
val trainSet = augment(trainG)
val model = LogisticRegressionWithSGD.train(trainSet, 10)
```

现在我们拥有了一个命名为 model 的逻辑回归模型。接下来呢？当然了，模型最重要部分的就是提供了函数 predict()，它接收特征向量作为输入并输出它认为最合适的标签。我们可以对训练集和测试集的一部分顶点进行逐一尝试。但一般而言，我们会一次性处理完所有的顶点。清单 7.8 中的函数 perf() 评估了算法的效果，具体而言是准确率指标，对比了人工打标的标签以及 model.predict() 产生的结果。

清单 7.8 评估逻辑回归模型的效果

```
import org.apache.spark.rdd.RDD

def perf(s:RDD[LabeledPoint]) = 100 * (s.count -
  s.map(x => math.abs(model.predict(x.features)-x.label)).reduce(_ + _)) /
  s.count

perf(trainSet)
res3: Double = 92.3076923076923
```

理想情况下，当用训练集来测试模型时，我们希望获得 100% 或相差不多的结果，但是 92% 的效果也不错。使用训练集上得到的模型再对训练集进行预测，这其实是一种作弊行为，但这也是一个说明我们的处理方法没有完全错误的明智做法。现在

我们使用测试数据来进行预测，如图 7.9 和清单 7.9 所示。

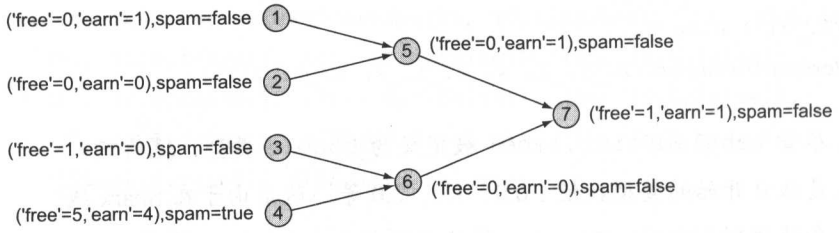


图 7.9 测试数据。

清单 7.9 在测试数据集上构建图并进行效果评估

```
val testV = sc.makeRDD(Array((1L, (0,1,false)), (2L, (0,0,false)),
  (3L, (1,0,false)), (4L, (5,4,true)), (5L, (0,1,false)),
  (6L, (0,0,false)), (7L, (1,1,true))))

val testE = sc.makeRDD(Array(Edge(1L,5L,""), Edge(2L,5L,""),
  Edge(3L,6L,""), Edge(4L,6L,""), Edge(5L,7L,""), Edge(6L,7L,"")))

perf(augment(Graph(testV,testE)))
res4: Double = 85.71428571428571
```

在测试数据集上，我们的模型效果是 86%。尽管和预期不一样，但效果也并不差。造成模型效果不是特别好的原因在于，训练数据集仅包含了少量的顶点（网页），采用的特征数也不多——应该构造上百个特征来对应上百个触发单词。一个较为理想的训练数据集至少应该包含 100,000 个网页样本。我们在训练集和测试集上都各预测错了一个标签。所以除了全部 100% 正确的情况外，我们不可能得到一个比以上结果更高的分数——这也是在这样一个小数据集上测试的另一个特性。

7.3.3 使用幂迭代聚类进行图像分割（计算机视觉）

在 7.3.1 节，我们介绍了文档聚类的方法。在本小节，我们会介绍如何使用无监督学习进行图像中像素的聚类。这种方法可以解决计算机视觉中的图像分割问题。主要思路是尝试给图像的每个像素赋予不同的标签。如图 7.10 所示，我们希望给左边图片中的每个像素打上“狮子”或者“背景”的标签。右边的图展示了一种可行的分割形式。通过对图像像素进行聚类，并为每个聚类赋予两种（或多种）颜色中

的一种，即可如图 7.10 一样对图像进行分割。

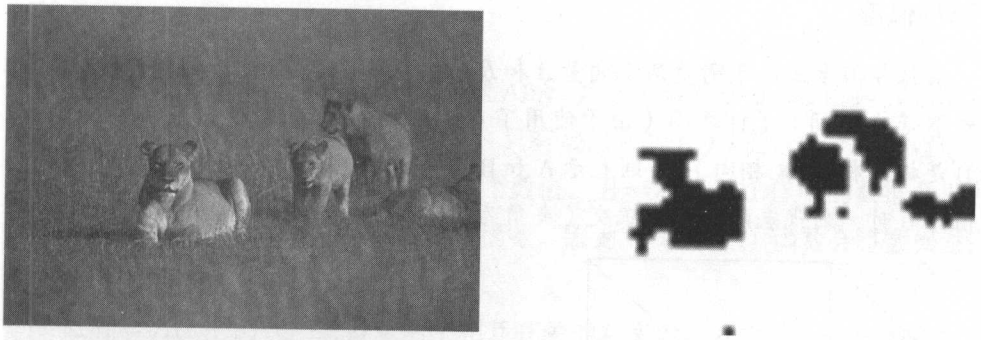


图 7.10 左边为原始图（本书为黑白印刷），右边为分割后的结果。低分辨率使得算法可以在单核的 4GB 机器上运行。聚类的个数需要提前设定，我们采用了 2 作为默认值。

对于聚类算法，每一个像素对应红 / 绿 / 蓝的立方体内的某一个位置，正如图 7.11 所示，聚类算法的目的是将这些像素聚成两个类。

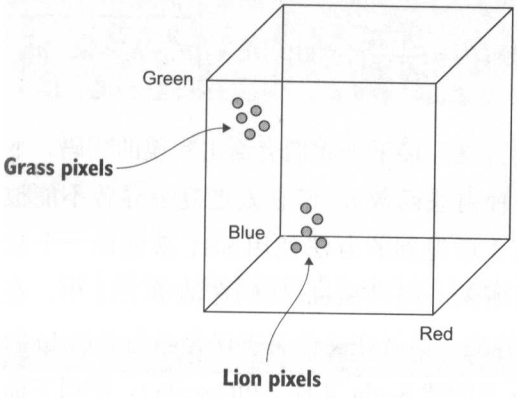


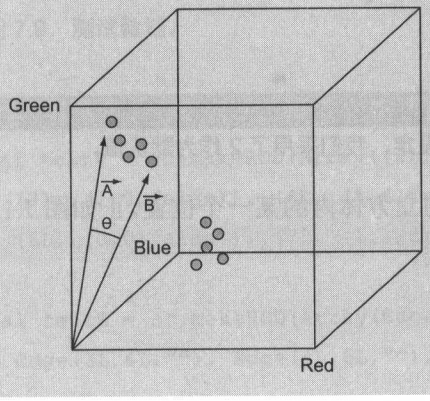
图 7.11 像素属于三维向量空间——每一个向量拥有三个数值（红、绿、蓝的强度），聚类算法的目标是找到两个聚类。

我们的聚类算法——幂迭代聚类（PIC），是属于 Spark MLlib 模块的，尽管它使用了 GraphX 形式的图作为输入。作为输入的图，要求顶点间是全连通的（每个顶点都和其他所有的顶点相连），边的权重则代表了两个顶点的“距离”。

在这个例子中，我们利用特征向量（红、绿、蓝），并使用余弦相似度作为相似度衡量方法。我们将每个像素点转换成长度为 3 的向量。Frank Lin 和 William Cohen 在 2010 年的论文 *Power Iteration Clustering* 里最先提出了 PIC 算法，他们也建议使用余弦相似度。

余弦相似度

余弦相似度是用于衡量两个向量 A 和 B 的相似度的方法之一。它有一个性质，就是取值在区间 [−1,1] 之间（由于使用了 cos(θ) 进行衡量而不是 θ），1 代表 A 和 B 是极度相似的（相同），−1 则表示 A 和 B 完全不相似（指向相反的方向）。同时，取值为 0 时，则代表两个向量正交（夹角为直角）。



$$\cos(\theta)=\frac{A\cdot B}{\|A\|\|B\|}，其中\ \|A\|=\sqrt{a_1^2+a_2^2+\ldots+a_n^2}$$

现在你已经找到如何寻找相似像素的方法，接下来我们会给出实现的代码。下面的代码中使用到了 lambda 表达式（一种内联函数），但它太过复杂导致不能被 Spark REPL 正确处理，因此我们按照第 3 章提到的方法使用 SBT 来编译一个独立的程序。相关的 SBT 文件如清单 7.10 所示，同时 Scala 代码在清单 7.11 中。在 Spark 的主目录下创建一个名为 PIC 的文件夹，并且在文件夹下保存名为 PIC.sbt 的 SBT 文件。然后在目录 PIC/src/main/scala 下创建 Scala 文件（PIC.scala）。使用下面的命令来编译和打包：

```
sbt package
```

如果执行成功，在目录 PIC/target/scala-2.10 下会生成相关的 jar 文件。然后就可以使用以下的命令来执行这个程序：

```
bin/spark-submit PIC/target/scala-2.10
```

代码要求输入文件（图片文件为 105053.jpg）放在相同的目录下，并会创建一个分割后的图片作为输出（out.png），如图 7.10 所示。图片文件是包含在伯克利分割数据集里的，可以通过以下方式来下载：

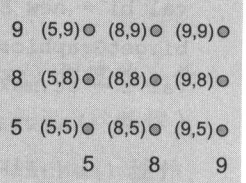
```
wget https://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/
    ➥ BSDS300-images.tgz
```

很多代码会使用到 Java 的图片处理 API，并会尽量减少 driver 的内存占用（更偏向使用集群中 worker 的内存）。例如，只有被放入到 RDD 后，红、绿、蓝的数据才会被从一个封装好的整型数中拆成多个数值。另一方面，RDD 中的数据会被分类封装在 Array 中，而不使用 `collect()` 函数，因为 `collect()` 函数会将数据发送回 driver。

但清单 7.11 中的代码隐藏着两个相互有关的新概念，其中的一个极为巧妙。代码 `r.cartesian(r)` 在 RDD `r` 上执行了笛卡儿积。它创建了一个由二元组(`Tuple2`)组成的 RDD，代表着 `r` 中元素所有可能的两两排列。

笛卡儿积

假设存在一个这样的 RDD，如 `val myRDD:RDD[Int]=sc.makeRDD(Array(5,8,9))`，具有三个元素，则 myRDD 的笛卡儿积就拥有 $3^2 = 9$ 个元素，笛卡儿积结果的 RDD 的类型是 `Tuple2[Int,Int]`，如右图所示。



由于需要计算所有像素两两间的相似度，所以我们使用了笛卡儿积来处理。

清单 7.10 图像分割的 PIC.sbt

```
scalaVersion := "2.10.5"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0"
libraryDependencies += "org.apache.spark" % "spark-graphx_2.10" % "1.6.0"
libraryDependencies += "org.apache.spark" % "spark-mllib_2.10" % "1.6.0"
```

清单 7.11 图像分割的 PIC.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.mllib.clustering.PowerIterationClustering
import org.apache.spark.graphx._

import java.awt.image.BufferedImage
```

```
import java.awt.image.DataBufferInt
import java.awt.Color
import java.io.File

import javax.imageio.ImageIO

object PIC {
  def main(args: Array[String]) {
    val sc = new SparkContext(new SparkConf().setMaster("local")
                                             .setAppName("PIC"))
    val im = ImageIO.read(new File("105053.jpg"))
    val ims = im.getScaledInstance(im.getWidth/8, im.getHeight/8,
                                   java.awt.Image.SCALE_AREA_AVERAGING)
    val width = ims.getWidth(null)
    val height = ims.getHeight(null)
    val bi = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB)
    bi.getGraphics.drawImage(ims, 0, 0, null)
    val r = sc.makeRDD(bi.getData.getDataBuffer
                        .asInstanceOf[DataBufferInt].getData)
              .zipWithIndex.cache
    val g = Graph.fromEdges(r.cartesian(r).cache.map(x => {
      def toVec(a:Tuple2[Int,Long]) = {
        val c = new Color(a._1)
        Array[Double](c.getRed, c.getGreen, c.getBlue)
      }
      def cosineSimilarity(u:Array[Double], v:Array[Double]) = {
        val d = Math.sqrt(u.map(a => a*a).sum * v.map(a => a*a).sum)
        if (d == 0.0) 0.0 else
        u.zip(v).map(a => a._1 * a._2).sum / d
      }
      Edge(x._1._2, x._2._2, cosineSimilarity(toVec(x._1), toVec(x._2)))
    }).filter(e => e.attr > 0.5), 0.0).cache
    val m = new PowerIterationClustering().run(g)
    val colors = Array(Color.white.getRGB, Color.black.getRGB)
    val bi2 = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB)
    m.assignments
```

```
      .map(a => (a.id/width, (a.id%width, colors(a.cluster))))
      .groupByKey
      .map(a => (a._1, a._2.toList.sortBy(_._1).map(_._2).toArray))
      .collect
      .foreach(x => bi2.setRGB(0, x._1.toInt, width, 1, x._2, 0, width))
    ImageIO.write(bi2, "PNG", new File("out.png"));
    sc.stop
  }
}
```

现在开始讲解最巧妙的部分了。我们使用 zipWithIndex() 来给像素点进行编号，方便追踪与使用。但我们注意到，在执行完 zipWithIndex() 后要再执行 cache() 操作。如果不这么做，则有可能最终 Spark 会执行两次 zipWithIndex()（由于 lazy 实现的原因），而第二次执行时则会产生不一样的编号！正如 Matei Zaharia 在 Jira 上 SPARK-3098 里的评论所说的，随机性是出于加速洗牌性能的考虑，所以并不能保证 RDD 内部元素的排序。

> Spark 小贴士：如果一个 RDD 是基于 zipWithIndex() 创建的，并会参与到自连接操作中（如与自己执行 join() 或 cartesian() 操作），那么可以在 zipWithIndex() 后、自连接操作前，附加上 cache() 操作，用于保证 RDD 中元素的顺序保持一致。

经过笛卡儿积处理后，我们得到了对应的代表边的 RDD，当然这个 RDD 的元素个数少于实际的边数，因为我们并不需要处理所有的边。我们仅保留相似度大于等于 0.5 的边。这样的做法除了可以加速 PIC 算法外，另一个原因是由于 PIC 算法要求所有的相似度必须大于等于 0，所以我们把不合格的边排除在外以免影响 PIC 算法的处理过程。

PIC 是一个通用的聚类算法。这里我们将它用于像素的聚类，以达到图像分割的最终目的。但 PIC 也可用于其他类型的聚类任务，只要提供了计算元素间相似度的方法。

7.4 穷人（简化版）的训练数据：基于图的半监督学习

迄今为止我们已经介绍过监督学习和无监督学习，接下来将会继续介绍融合了

两者优点的半监督学习。尽管监督学习的优势之一是可以产生人类可理解的标签(因为它是由打标的数据训练得来的)，但它的缺点也很明显：需要耗费人力对所有的数据进行打标。这是一笔昂贵的开销。而由于无监督学习并不要求训练数据是打标的，所以很容易获得大规模的训练数据来进行处理。

半监督学习背后的思路，一般是首先在无标的数据上进行无监督学习。这提供了一些可用于进行数据打标的数据结构信息。然后基于这些增强的打标数据，我们运行监督学习来获得更有效的模型。

在本节中，我们需要处理多维空间的数据问题，例如二维平面、三维立方或者更高维度的情况。这些空间中的坐标轴可以代表任意的变量：气温、测试分数、人口等。假设基于相似的点会被聚集在一起，我们会对这些空间的点附上相关的标签。

例如，对于有线电视和因特网的提供商而言，他们所关心的二维平面的坐标轴，分别表示看电视的时长和数据传输的千兆字节数。在这种情况下，我们可以区分出不同的用户类型，如爱好电视的用户、爱好因特网的用户，以及两方面均爱好的用户。

在构建一个图来适应这些数据点的分布时，我们可以确定不同类型用户群体的边界。事实证明，在对新用户进行处理时，构建这样一个图来确定相似用户群体，可以有效帮助预测算法进行预测。

在实现上述思路时，我们首先实现了一个K近邻图构建算法（与进行预测的K近邻算法做好区分，它们是不一样的算法，本书也不会涉及K近邻算法），作为我们的无监督学习算法，并将它应用于大部分是无标数据的数据集上。然后我们实现了一个简单的标签传播算法，它会将标签传递给附近的无标顶点上。最后，我们实现了一个简单的`knnPredict()`函数来预测新数据点的标签。

> **注意**：有许多机器学习算法都是以字母K开头的。一般而言，这指的是模型中一个通常会被命名为K的参数，该参数需要由用户进行设置。K的实际意义与具体的算法有关，尽管在一部分算法中它们的意义是相近的。例如，在聚类算法K-Means和K-Medians中，K指的都是我们所期望的聚类数。而在K近邻算法中，K是指一个顶点应该与多少个相似的顶点进行计算。在我们的例子中，我们同样需要设置K的值，它代表多少个相似的点。

图7.12展示了开始的状态，图7.13则展示了经过K近邻和半监督学习标签传播算法后的状态。这些数据的分布类似马蹄铁的形状，是K-Means算法不能处理的

经典反例（K-Means 算法与我们的 K 近邻算法没有任何关联）。K-Means 算法的主要思路是寻找聚类的中心点，因此在处理距离长、链状连接的点集时效果不好。但由于我们的方法使用了 K 近邻算法进行图构建，可以很好地跟踪这些长链的情况，因此可以得到较好的结果。

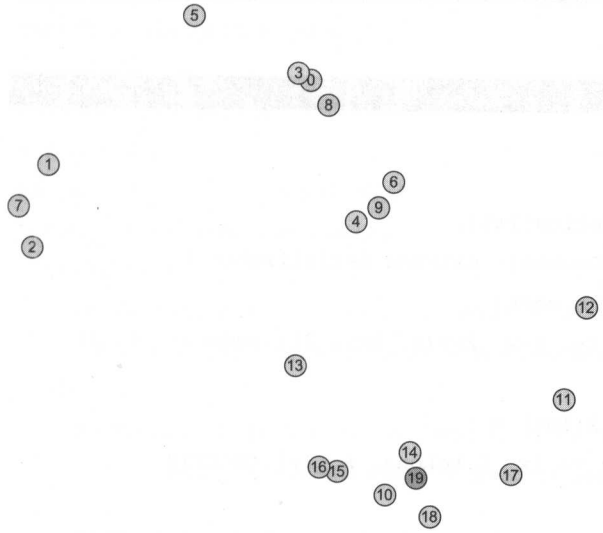


图 7.12 开始状态：处于二维平面的点集，除了两个有标的数据外，其他绝大部分数据都是无标的。

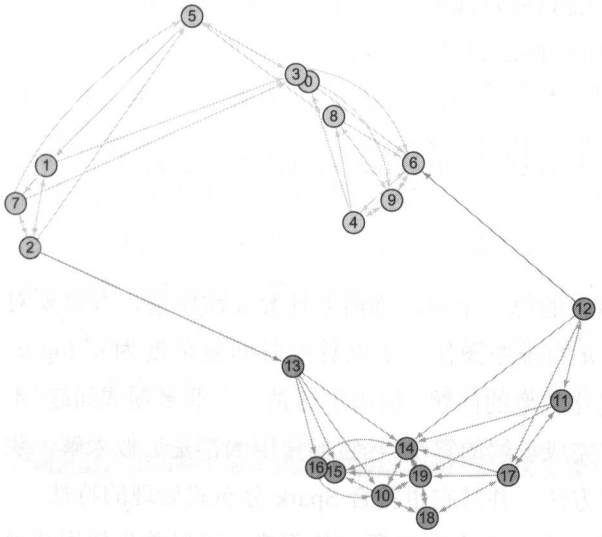


图 7.13 经过 K 近邻图构建算法和半监督学习标签传播算法后的状态。

7.4.1 K近邻图构建

Spark 1.6版本没有包含K近邻算法，在Jira的SPARK-2335中可以找到相关内容。

理论上，寻找近邻点的操作不会太复杂。对于每个点而言，从所有的点中找到与它相近的 K 个点，然后在它们之间添加上边。这种简单的、暴力的方法如清单7.12所示。

清单7.12 暴力搜索K近邻

```
import org.apache.spark.graphx._

case class knnVertex(classNum:Option[Int],
                     pos:Array[Double]) extends Serializable {
  def dist(that:knnVertex) = math.sqrt(
    pos.zip(that.pos).map(x => (x._1-x._2)*(x._1-x._2)).reduce(_ + _))
}
def knnGraph(a:Seq[knnVertex], k:Int) = {
  val a2 = a.zipWithIndex.map(x => (x._2.toLong, x._1)).toArray
  val v = sc.makeRDD(a2)
  val e = v.map(v1 => (v1._1, a2.map(v2 => (v2._1, v1._2.dist(v2._2)))
                                .sortWith((e,f) => e._2 < f._2)
                                .slice(1,k+1)
                                .map(_._1)))
           .flatMap(x => x._2.map(vid2 =>
            Edge(x._1, vid2,
                 1 / (1+a2(vid2.toInt)._2.dist(a2(x._1.toInt)._2)))))
  Graph(v,e)
}
```

关键问题在于性能。对 n 个点中的每一个点，都需要计算 n 次距离，并需要对这些距离进行时间复杂度为 $n \log n$ 的排序操作。所以总的时间复杂度为 $n^2 \log n$。所有K近邻算法的变种都会尝试优化效率的问题。但由于这是一个非多项式问题(不存在可以在任意合理的时间范围内完成求解的算法)，它们使用的都是近似求解。我们接下来会介绍这样一种近似求解方法，并且它也符合Spark分布式处理的特性。

但首先请查看清单7.12。若需要执行清单7.12所示的程序，可以首先使用清单7.13的程序来生成如图7.14所示的数据集。清单7.14则用于将数据转成Gephi需要

的 .gexf 文件形式，对我们的 knnVertex 函数输出的颜色和位置标签进行转换。清单 7.15 执行了算法并输出了 .gexf 文件。在这里我们使用了 k=4 来选择近邻顶点数。3 和 4 都是 k 的典型取值。

清单 7.13 生成样本数据

```
import scala.util.Random

Random.setSeed(17L)
val n = 10
val a = (1 to n*2).map(i => {
  val x = Random.nextDouble;
  if (i <= n)
    knnVertex(if (i % n == 0) Some(0) else None, Array(x*50,
      20 + (math.sin(x*math.Pi) + Random.nextDouble / 2) * 25))
  else
    knnVertex(if (i % n == 0) Some(1) else None, Array(x*50 + 25,
      30 - (math.sin(x*math.Pi) + Random.nextDouble / 2) * 25))
})
```

清单 7.12 中程序的核心是对 RDD 中的边进行计算，我们会将它作为参数传递给 Graph() 进行图的创建。计算过程的时间复杂度是 n^2，包含一个外部的 RDD.map() 操作（在变量 v 上操作，v 为顶点的 RDD）和一个内部的 Array.map() 操作（在变量 a2 上操作，a2 为数组存储形式的 v）。对于每个顶点，我们计算所有的距离并将它们进行排序，从中选择 k 个最短的距离（我们忽略下标为 0 的顶点，因为它代表与自身的距离，即距离为 0）。当我们最后构建实际的边集 Ege() 时，使用 Edge 的属性来存储距离的倒数。这个值用于半监督学习的标签传播算法中，但 K 近邻算法本身不需要使用它。

清单 7.12 中的 flatMap() 是一种高效的做法：它同时高效地完成了 map()（将点集转换成边集）和 flatten()（将多个边集合并成边集的集合）两个操作的功能。

清单 7.14 将基于 knnVertex 的图定制化输出为（带布局的）Gephi 的 .gexf 文件

```
import java.awt.Color
def toGexfWithViz(g:Graph[knnVertex,Double], scale:Double) = {
  val colors = Array(Color.red, Color.blue, Color.yellow, Color.pink,
```

```
                    Color.magenta, Color.green, Color.darkGray)
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
  "<gexf xmlns=\"http://www.gexf.net/1.2draft\" " +
        "xmlns:viz=\"http://www.gexf.net/1.1draft/viz\" " +
        "version=\"1.2\">\n" +
  " <graph mode=\"static\" defaultedgetype=\"directed\">\n" +
  "   <nodes>\n" +
  g.vertices.map(v =>
    "     <node id=\"" + v._1 + "\" label=\"" + v._1 + "\">\n" +
    "       <viz:position x=\"" + v._2.pos(0) * scale +
           "\" y=\"" + v._2.pos(1) * scale + "\" />\n" +
    (if (v._2.classNum.isDefined)
      "       <viz:color r=\"" + colors(v._2.classNum.get).getRed +
            "\" g=\"" + colors(v._2.classNum.get).getGreen +
            "\" b=\"" + colors(v._2.classNum.get).getBlue + "\" />\n"
     else "") +
    "     </node>\n").collect.mkString +
  "   </nodes>\n" +
  "   <edges>\n" +
  g.edges.map(e => "     <edge source=\"" + e.srcId +
                "\" target=\"" + e.dstId + "\" label=\"" + e.attr +
                "\" />\n").collect.mkString +
  "   </edges>\n" +
  " </graph>\n" +
  "</gexf>"
}
```

清单 7.15 在样本数据上运行 K 近邻算法并输出为 .gexf 文件

```
val g = knnGraph(a, 4)

val pw = new java.io.PrintWriter("knn.gexf")
pw.write(toGexfWithViz(g,10))
pw.close
```

分布式 K 近邻算法

在众多的近似 K 近邻图构建算法中，大部分都是基于传统的串行化处理方式，

而不支持并行化处理。一个创新的并行化实现来自于 2012 年由 Wang 等人完成的微软研究文章 *Scalable k-NN graph construction for visual descriptors*。

这篇文章介绍了分布式计算的大量优化方法，但这里我们仅实现其中的一种。这对他们的论文不是很客观，但可以给我们带来实用的 Spark 上实现 K 近邻算法的思路。

Wang 的文章里一个关键思想（这也是唯一被我们采用的）是，将空间划分成网格，并在网格的各个单元上执行暴力的 K 近邻图构建算法。在图 7.14 中可以看到，空间被划分成了多种 3×3 的网格（在第一种划分中，最后一个格子的宽和高都为 0）。当我们取 m=3 时，c 代表希望使用的不同网格分割的个数，则复杂度为 $cm^2(n/m^2)(n/m^2)\log(n/m^2) = c(n/m)^2\log(n/m^2)$。

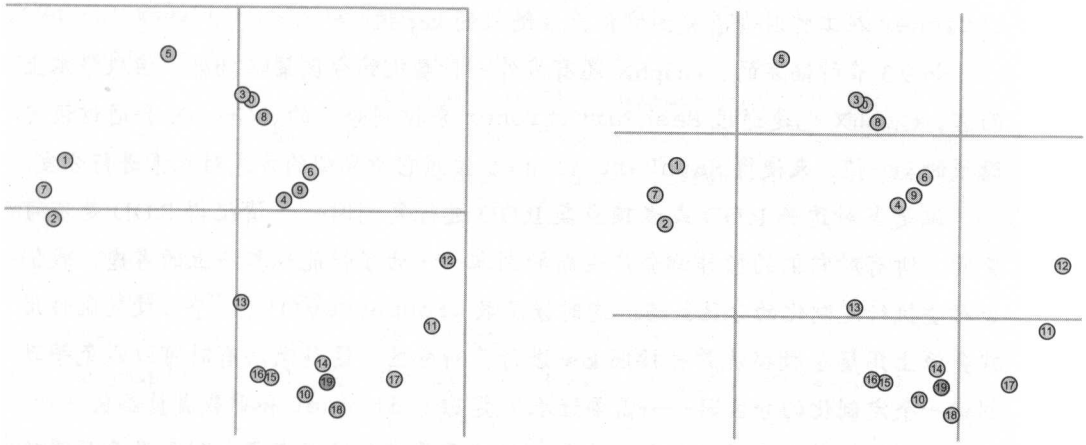


图 7.14 分布式 K 近邻图的构建。将空间划分成网格，并在网格的各个单元上执行暴力的 K 近邻图构建算法。为了避免有的边会跨过多个单元，改变网格分布后再次执行，将两次生成的边集进行合并。

这是我们采用的一种简单策略。Wang 的论文里的完整算法比较复杂，如使用到主成分分析法（PCA）来决定网格的方向（将空间分割成平行四边形而不是正方形），使用很多随机的网格，并将有向边转换成无向边。

在 Spark 的实现中，我们希望将每个网格单元分配到不同的执行单元（任务）。这可通过设置 Spark RDD 的分区策略来实现。

清单 7.16 使用 `groupByKey()` 来对数据进行洗牌，将数据分配到不同的分区中，如图 7.14 所示，然后使用 `mapPartitions()` 来在每个单元内进行暴力的 K 近邻边的生成。函数 `mapPartitions()` 允许我们获得网格单元内的所有顶点子集（保

存在变量 af 内），假设这个网格单元里一共有 d 个顶点，则经过 d^2 次距离的计算后，我们可以完成 K 近邻顶点的边生成。

RDD 分区策略（和 mapPartitions() 函数）

从基本层面上讲，RDD 是分布式的数据集，而 Spark 会根据 RDD 的分区器（partitioner）策略将数据分发到集群中的不同顶点。默认的分区器是 HashPartitioner，它将 key-value 形式的 Tuple 2[k,v] 按照 key 进行分发，将具有相同 key 的 RDD 数据元素分发到相同的顶点上。这种做法假设了 RDD 一开始的时候就以 key-value 的形式存储（如 PairRDDFunctions 的操作结果）。但如果要处理的是无格式的、老式的 RDD，它没有指定的 key，那么 Spark HashPartitioner 在工作时会首先为它们产生随机的 key 值。

如 9.3 节所描述的，GraphX 还有另外一种层次的分区策略抽象。但从根本上而言，GraphX 也是通过 HashPartitioner 来控制分区的。GraphX 会通过设定隐藏的 key 值，来使得 HashPartitioner 按照它所期望的方式对元素进行分发。

但是当对边集 RDD 或者顶点集 RDD 进行复制时——请记得 RDD 是不可变的，所有对它们的操作都会产生新的副本——为了性能或算法上的考虑，我们会希望执行定制化的分区策略。这时候函数 groupByKey() 的一个方便的副作用就会派上用场：数据洗牌并按照 key 进行重新分区。这种做法有时可以避免手动创建一个定制化的分区器——需要继承父类 Partitioner 和重载成员函数。

分区的操作会在幕后进行，我们一般不需要进行过多考虑。但如果希望明确指定数据的存储方式，不论是出于性能还是算法上的考虑，这时我们才需要进行特殊处理。进行分区时一个重要的方法是使用 mapPartitions() 函数。

mapPartitions() 函数让你以 Scala 集合的形式在一个分区内对所有数据进行处理。你可以在每个执行单元处理分派给它的部分 RDD 数据前，进行一些耗时的设置或数据拆分工作，例如创建一个数据库游标或者初始化一个分析器。如果采用 RDD 的 map() 函数进行上述操作，它会在处理每一份数据时都需要重新执行这些耗时的操作，而不是在每个分区上仅执行一次。

我们会在样本数据上首先执行上述近似求解的 K 近邻图构建算法，然后执行下一节会介绍的半监督学习标签传播算法，结果如图 7.15 所示。

清单 7.16 分布式近似求解的 K 近邻图生成

```
def knnGraphApprox(a:Seq[knnVertex], k:Int) = {
  val a2 = a.zipWithIndex.map(x => (x._2.toLong, x._1)).toArray
  val v = sc.makeRDD(a2)
  val n = 3
  val minMax =
    v.map(x => (x._2.pos(0), x._2.pos(0), x._2.pos(1), x._2.pos(1)))
     .reduce((a,b) => (math.min(a._1,b._1), math.max(a._2,b._2),
                       math.min(a._3,b._3), math.max(a._4,b._4)))
  val xRange = minMax._2 - minMax._1
  val yRange = minMax._4 - minMax._3

  def calcEdges(offset: Double) =
    v.map(x => (math.floor((x._2.pos(0) - minMax._1)
                           / xRange * (n-1) + offset) * n
                + math.floor((x._2.pos(1) - minMax._3)
                             / yRange * (n-1) + offset),
                x))
    .groupByKey(n*n)
    .mapPartitions(ap => {
      val af = ap.flatMap(_._2).toList
      af.map(v1 => (v1._1, af.map(v2 => (v2._1, v1._2.dist(v2._2)))
                             .toArray
                             .sortWith((e,f) => e._2 < f._2)
                             .slice(1,k+1)
                             .map(_._1)))
          .flatMap(x => x._2.map(vid2 => Edge(x._1, vid2,
            1 / (1+a2(vid2.toInt)._2.dist(a2(x._1.toInt)._2)))))
          .iterator
    })

  val e = calcEdges(0.0).union(calcEdges(0.5))
                        .distinct
                        .map(x => (x.srcId,x))
                        .groupByKey
                        .map(x => x._2.toArray
```

```
                                .sortWith((e,f) => e.attr > f.attr)
                                .take(k))
                        .flatMap(x => x)
  Graph(v,e)
}
```

Spark 小贴士：RDD 函数 union() 与 SQL 的 UNION 操作不一样，不会自动删除重复的数据。若需要获得不重复的 RDD 数据，你需要手动触发 distinct() 操作。

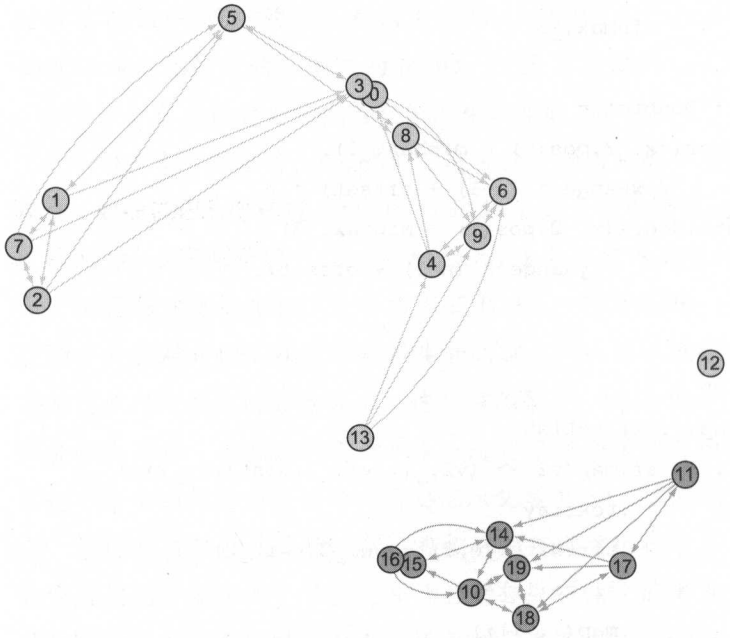


图 7.15 执行近似求解的分布式 K 近邻图构建算法，和下一节要介绍的半监督学习标签传播算法后，所得的结果示意图。里面仅包含了三分之二的边。与图 7.13 中的实际结果不一样，顶点 13 被误分类了，顶点 12 则不属于任何类。但采用近似求解带来的好处是算法可以分布式执行，适用于大型图。

请注意，为了让 groupByKey() 按我们期望的方式进行数据分区和洗牌，我们需要设置它的可选参数来指定分区的数目。否则，groupByKey() 会将一些较小的分区合并在一起。由于这会影响到算法效果，这种情况下我们仍希望有大量的不同分区。我们将这个参数设为可能出现的最大取值（n*n），但实际的分区数有可能会少于这个最大值（因为会有空的网格单元），groupByKey() 会使用尽可能多的非

空分区。

也请注意，正如之前提到的，我们将函数传递进来的参数提供给 mapPartitions() 操作，但这些数据严格意义上不是单一键值的形式，它们是一个包含了多个键值的集合。因为我们假设会给每个分区提供单一的键值，所以首先对这些数据执行 flatMap() 操作，以减少额外的嵌套。

在最终计算边集 e 时，我们使用 union() 将图 7.14 所示的两个网格单元的不同边集合并起来。由于这会导致与每个顶点关联的边的条数超过 k，我们使用了 groupByKey()、map()、flatMap() 这一系列操作来进行裁剪。函数 sortWith() 内使用的比较器是“大于”，而不是默认的“小于”，因为边属性上存储的是距离的倒数，而不是距离本身。

7.4.2 半监督学习标签传播算法

迄今为止我们完成了从数据集的所有点中提取出它们的结构信息的操作，还没有考虑应该为它们打上何种标签。图 7.16 展示了我们要创建的图结构。顶点 9 和顶点 19 是有标签的，其他点都没有标签。我们会通过一个标签传播算法来为其他顶点打上标签，然后会介绍如何使用这个完全打好标签的模型来对新数据进行标签预测。

现在我们开始实现标签传播算法，具体代码如清单 7.17 所示。我们在第 5 章时讨论过 Spark 内置的标签传播算法，介绍了如何基于已打标的数据集，通过标签收敛的过程来识别社区。

相比之下，我们在本章使用的方法，是基于部分已知标签的顶点和在无监督学习过程中构建的图结构，将标签传播到更多部分的无标顶点上。我们的方法也会考虑边的距离，邻近点的权重会更大，所以我们的算法大部分情况下都会收敛。

算法过程如下：

1. 对于每条与有标顶点关联的边，将有标顶点的标签和相应的权重值（边距离的倒数）同时传给边的起始点和目标点。
2. 对于每个顶点，分别（按标签划分）进行分数的累加。如果该顶点的标签不是预先确定的，则进行标签修改，即将总分高的标签作为顶点的新标签。
3. 如果没有顶点的标签发生变动，或者达到最大的迭代次数，结束。

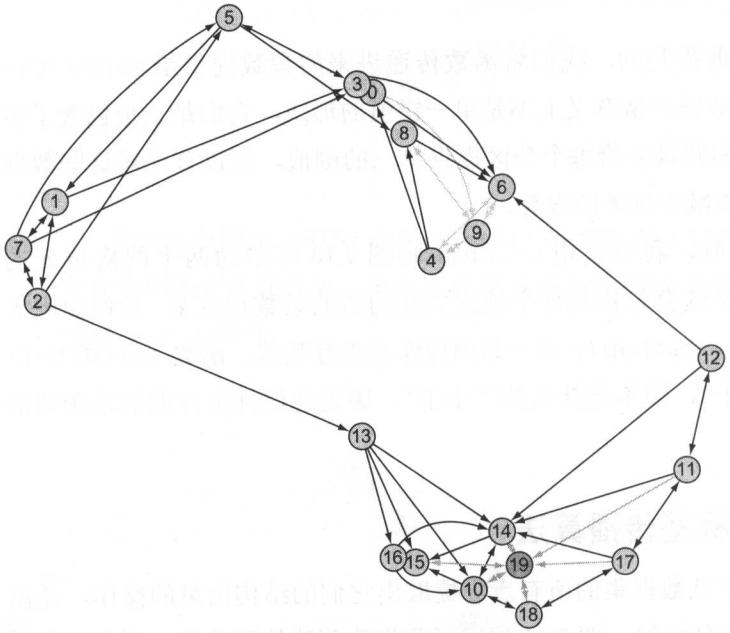


图 7.16 经过分布式的、近似求解的无监督学习算法处理后的图。现在仅有基本的结构，但无标的顶点仍未被打上相应的标签。

我们使用 aggregateMessages()（和 joinVertices()）来完成这项工作，而不使用 Pregel() 函数。因为 Pregel() 的终止条件是不再有信息的发送。但在我们的处理中，我们会一直发送有标顶点的标签，来保证有标的顶点可以获得自身的标签。因此，使用 Pregel() 并不适合。

请注意以上算法的一个重点是，它将图当作无向图来处理。实际的处理过程会有轻微的不同，它会保证预先已确定标签的顶点永远不会改变自身的标签，但总体而言标签可以沿着边的任意方向进行传播。

图 7.17 举例说明了算法在图 7.12 和图 7.13 所得的 K 近邻图上的迭代过程。

Scala 小贴士：操作算子 -> 是 Scala HashMap 获得键值对的一个方便操作。如果你熟悉 PHP 语言，这和 PHP 使用 => 进行数组初始化是类似的。

清单 7.17 半监督学习标签传播算法

```
import scala.collection.mutable.HashMap
def semiSupervisedLabelPropagation(g:Graph[knnVertex,Double],
                                   maxIterations:Int = 0) = {
```

```
val maxIter = if (maxIterations == 0) g.vertices.count / 2
              else maxIterations

var g2 = g.mapVertices((vid,vd) => (vd.classNum.isDefined, vd))
var isChanged = true
var i = 0

do {
  val newV =
    g2.aggregateMessages[Tuple2[Option[Int],HashMap[Int,Double]]](
      ctx => {
        ctx.sendToSrc((ctx.srcAttr._2.classNum,
                       if (ctx.dstAttr._2.classNum.isDefined)
                          HashMap(ctx.dstAttr._2.classNum.get->ctx.attr)
                       else
                          HashMap[Int,Double]()))
        if (ctx.srcAttr._2.classNum.isDefined)
          ctx.sendToDst((None,
                         HashMap(ctx.srcAttr._2.classNum.get->ctx.attr)))
      },
      (a1, a2) => {
        if (a1._1.isDefined)
          (a1._1, HashMap[Int,Double]())
        else if (a2._1.isDefined)
          (a2._1, HashMap[Int,Double]())
        else
          (None, a1._2 ++ a2._2.map{
            case (k,v) => k -> (v + a1._2.getOrElse(k,0.0)) })
      }
    )

  val newVClassVoted = newV.map(x => (x._1,
    if (x._2._1.isDefined)
      x._2._1
    else if (x._2._2.size > 0)
      Some(x._2._2.toArray.sortWith((a,b) => a._2 > b._2)(0)._1)
```

```
      else None
    ))

    isChanged = g2.vertices.join(newVClassVoted)
                           .map(x => x._2._1._2.classNum != x._2._2)
                           .reduce(_ || _)

    g2 = g2.joinVertices(newVClassVoted)((vid, vd1, u) =>
      (vd1._1, knnVertex(u, vd1._2.pos)))
    i += 1
  } while (i < maxIter && isChanged)

  g2.mapVertices((vid,vd) => vd._2)
}
```

预测

现在我们获得了经过半监督学习后的图，可以使用它来“预测”标签了。给定一个坐标为（x，y）的点，它的分类（标签）应该是什么呢？清单7.18包含了一个极其简单的预测函数。它找到最近的有标的顶点（不论这个标签是预先确定的还是在传播过程中确定的），然后返回该有标顶点的标签。技术上而言，这是k=1时的K近邻预测方法（注意与K近邻图构建算法做区分）。

清单7.19展示了如何在清单7.17的semiSupervisedLabelPropagation()产生的模型上来调用这个简单的knnPredict()函数。

清单7.18 基于半监督学习所得的图的预测函数

```
def knnPredict[E](g:Graph[knnVertex,E],pos:Array[Double]) =
  g.vertices
   .filter(_._2.classNum.isDefined)
   .map(x => (x._2.classNum.get, x._2.dist(knnVertex(None,pos))))
   .min()(new Ordering[Tuple2[Int,Double]] {
    override def compare(a:Tuple2[Int,Double],
                         b:Tuple2[Int,Double]): Int =
      a._2.compare(b._2)
    })
  ._1
```

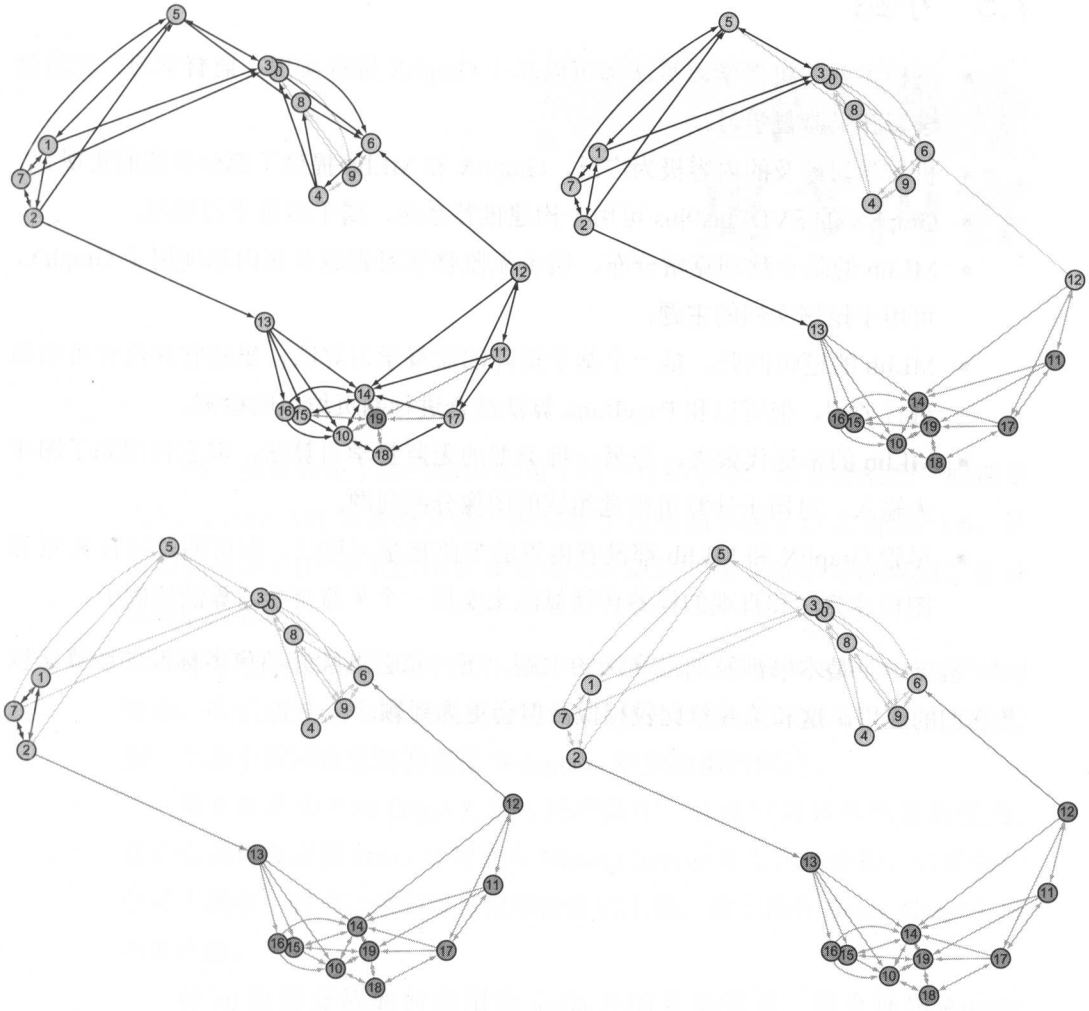


图 7.17 在 K 近邻图上，半监督学习标签传播算法的迭代过程。

清单 7.19 运行半监督学习标签传播算法，并使用它对坐标为（30.0，30.0）的点进行类别（标签）的预测

```
val gs = semiSupervisedLabelPropagation(g)
knnPredict(gs, Array(30.0,30.0))
res5: Int = 0
```

7.5 小结

- 三种主要的机器学习算法都可以基于 GraphX 进行实现：监督学习、无监督学习和半监督学习。
- 机器学习涉及的内容极为广泛，GraphX 和 MLlib 提供了部分算法的实现。
- GraphX 的 SVDPlusPlus 可用于构建推荐系统，属于监督学习领域。
- MLlib 的隐含狄利克雷分布，属于无监督学习领域并在内部使用了 GraphX，可用于挖掘文档的主题。
- MLlib 的逻辑回归，是一个基于矩阵的监督学习算法，虽然它并没有用到图进行处理，但可以和 PageRank 算法结合进行网页垃圾的检测。
- MLlib 的幂迭代聚类，是另一种类型的无监督学习算法，但它使用到了图作为输入，可用于计算机视觉领域的图像分割问题。
- 尽管 GraphX 和 MLlib 都没有内置的半监督学习算法，但可通过结合 K 近邻图构建算法和直观的标签传播算法来实现一个半监督学习算法的例子。

机器学习是本书涉及的最高深的主题。下一章会继续介绍更多标准的图算法以及它们的应用。这和第 6 章比较相似，但会更为新颖。

第3部分
更多内容

第 3 部分补充了缺失的内容和文档。在第 8 章，你会接触到一些期望成为 GraphX API 的部分算法，但事实上现在仍不属于官方 Spark 1.6。从读取标准的 RDF 格式的图数据到图合并算法，第 8 章的算法填充了部分缺失的内容。

第 8 章还提及了如何使用 IndexedRDD，这是类似 RDD 的 HashMap 版本。我们通过一个例子来展示它是如何提升性能的。最后，你还会接触到一个基于图同构思路的查找 Wikipedia 缺失数据的例子。

第 9 章是关于将 GraphX 投入到产品中，并进行调试和性能调优的。我们会向你介绍像 DAG 可视化和 History Server 等工具的使用，并提供一些用于缓存、断点记录和串行化等操作的工具，用于提升 Spark GraphX 应用的性能。

第 10 章会介绍如何使用除 Scala 外的其他语言。组合使用 Apache Zeppelin 和 d3.js，可以在笔记本交互式软件中进行内建的图可视化操作，通过交互式脚本来展示图。不需要写入独立的文件并可以立即展示图，这种方式很强大，我们不需要额外微调渲染的参数。我们还提供了如何使用 Spark Job Server 的例子，它允许我们将 GraphX 包装成一个图数据库而不是图处理系统。最后，GitHub 上的 GraphFrames 提供了方便和高性能的通过 SQL 和 Cypher 语言进行图查询的方式。

8 缺失的算法

本章要点

- 读取 RDF 文件
- 图合并
- 过滤孤立顶点
- 使用 IndexedRDD 提升性能
- 简单的查找图同构的方法
- 计算全局聚类系数

在之前的章节里我们介绍了从边列表文件中读取图数据的方法。RDF 是另一种重要的被广泛应用的文件格式。本章会介绍如何读取这种格式的文件，并应用到 YAG03 数据集上。

除了第 6 章介绍的经典的图算法外，还存在一些更新颖的用于图数据库或图处理系统的算法。但部分算法还有待实现（在官方 Apache Spark 1.6 版本或者在 spark-packages.org 中并不可用）。

在本章中，我们会介绍如何实现上述部分算法。同时你会明白如何使用 IndexedRDD 来提升性能。IndexedRDD 最初是由 GraphX 代码主要贡献者的其中一员实现的，但没有被合并进 Apache Spark 的官方版本中。

8.1 缺失的基本图操作

本节的实现会依赖于下一节的读取 RDF 文件的相关内容。在介绍如何读取 RDF 文件前，我们首先需要掌握部分 GraphX（在 Spark 1.6 版本中）没有提供的图操作方法。第一个要掌握的图操作方法是通用的子图抽取，第二个则是对两个图进行合并。

8.1.1 通用意义上的子图

在 5.2 节的补充资料中介绍 GraphX 的 `subgraph()` 操作时，我们提到删除一个顶点的所有边时会留下一个孤立顶点，这会带来极大的不便，如图 8.1 所示。一般情况下，你会知道哪些边该保留、哪些边该去除，但并不了解哪些顶点需要被保留下来。

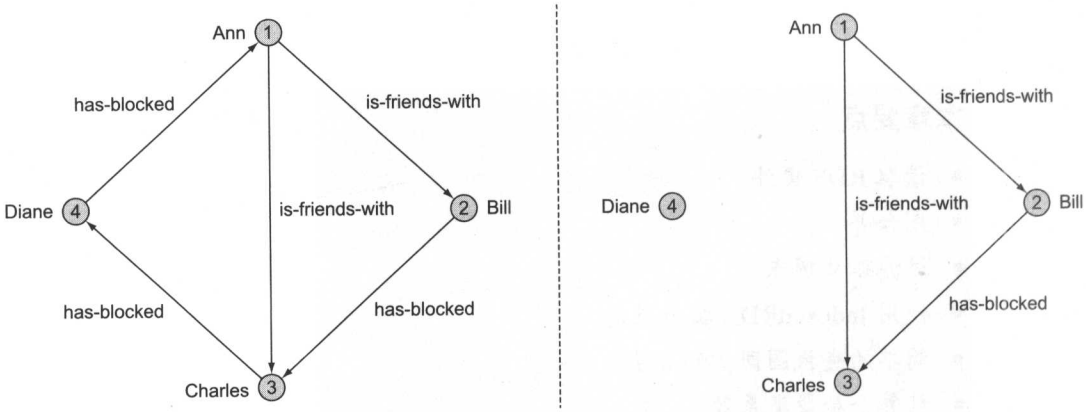


图 8.1 当使用 GraphX 的 subgraph() 来保留 is-friends-with 关系的边时，它会留下右边如 Diane 这样的离群点。

在执行 `subgraph()` 进行边过滤后，清单 8.1 中的代码提供了删除所有单独（孤立）顶点的实现方法。

清单 8.1 删除孤立顶点：在执行 subgraph() 后使用

```
import scala.reflect.ClassTag

def removeSingletons[VD:ClassTag,ED:ClassTag](g:Graph[VD,ED]) =
  Graph(g.triplets.map(et => (et.srcId,et.srcAttr))
                  .union(g.triplets.map(et => (et.dstId,et.dstAttr)))
                  .distinct,
        g.edges)
```

为了删除孤立顶点，清单 8.1 内的代码思路基于以下事实：函数 triplets() 返回的点集包含的都是有边的点。首先通过 triplets() 获得所有起始点，再调用 triplets() 获得所有的目标点，并通过 union() 操作将它们合并起来。然后在顶点集上执行 distinct() 操作后，我们就获得了一个用上述顶点集和原始边集构成的新图。

清单 8.2 包含了可用于图 8.1 的函数。输出首先展示了原始子图的情况，包括孤立顶点 Diane，然后展示了运行 removeSingletons 操作后的顶点集。

清单 8.2 函数 removeSingletons 的使用示例

```
val vertices = sc.makeRDD(Seq(
(1L, "Ann"), (2L, "Bill"), (3L, "Charles"), (4L, "Dianne")))
val edges = sc.makeRDD(Seq(
Edge(1L,2L, "is-friends-with"),Edge(1L,3L, "is-friends-with"),
Edge(4L,1L, "has-blocked"),Edge(2L,3L, "has-blocked"),
Edge(3L,4L, "has-blocked")))
val originalGraph = Graph(vertices, edges)
val subgraph = originalGraph.subgraph(et => et.attr == "is-friends-with")

// 展示子图中的顶点，包括 Dianne
subgraph.vertices.foreach(println)

// 调用 removeSingletons 函数，并展示结果对应的顶点
removeSingletons(subgraph).vertices.foreach(println)
```

原始图和原始子图的点集，包括点 Diane，如下所示：

```
(4,Dianne)
(3,Charles)
(2,Bill)
(1,Ann)
```

运行 removeSingletons 操作后，结果如下：

```
(3,Charles)
(2,Bill)
(1,Ann)
```

8.1.2 图合并

如果你有两个顶点类型均为 String 的图，有时候会需要对它们具有相同名称

的顶点进行合并，正如图 8.2 所示。合并意味着将两幅图中具有同样名称的顶点当作是同一个顶点，并将相毗邻的边合并在一起。新图中也会包含两幅图中其他没有重复的顶点和边。当两个图是从不同的数据来源构造出来的，并希望从中获得一个更大的图时，图合并的操作相当有用。

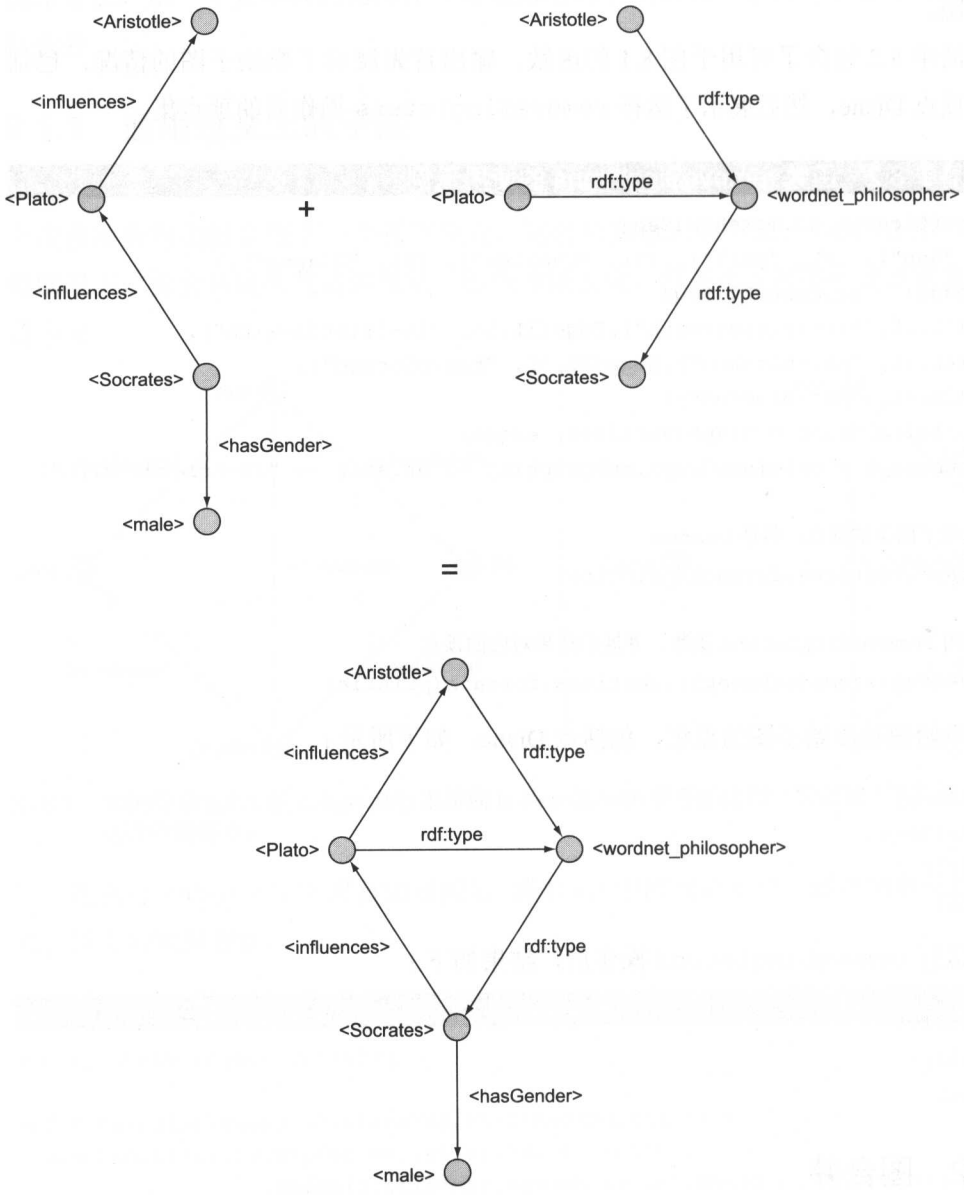


图 8.2 将两幅图中具有相同名称的点合并在一起，虽然这是一个相当实用的操作，但它并没有被包含在 GraphX（Spark 1.6 版本）中。

清单 8.3 中的代码展示了实现的逻辑。函数使用我们希望合并的两个图作为输入，并输出所有共有顶点合并后形成的单一图。

清单 8.3 将两个图合并成一个

```
import org.apache.spark.graphx._

def mergeGraphs(g1:Graph[String,String], g2:Graph[String,String]) = {
  val v = g1.vertices.map(_._2).union(g2.vertices.map(_._2)).distinct
                     .zipWithIndex
  def edgesWithNewVertexIds(g:Graph[String,String]) =
    g.triplets
     .map(et => (et.srcAttr, (et.attr,et.dstAttr)))
     .join(v)
     .map(x => (x._2._1._2, (x._2._2,x._2._1._1)))
     .join(v)
     .map(x => new Edge(x._2._1._1,x._2._2,x._2._1._2))
  Graph(v.map(_.swap),
        edgesWithNewVertexIds(g1).union(edgesWithNewVertexIds(g2)))
}
```

首先，我们构建了一个共用的顶点字典 v。接着使用第一个输入图生成了一个包含顶点属性值的 RDD，是通过代码 g1.vertices.map(_._2) 来实现的。对第二个输入图也执行同样的操作。我们可以通过 union 操作将两个 RDD 合并起来，并通过 distinct 操作来生成一个顶点属性值不重复的 RDD。最后，我们通过 zipWithIndex 操作来为每个顶点生成新的顶点 ID。对于图 8.2 进行处理，示例结果如下：

```
(Plato, 0)
(Aristotle, 1)
(wordnet_philosophers, 2)
(Socrates, 3)
(male,4)
```

嵌套函数 edgesWithNewVertexIds() 将图的边集和 v 内的 vertextIds 映射起来，在输入图 g1 和 g2 上都执行了相同的操作。最终的返回结果是一个对图 g1 和 g2 的边都进行了映射的新图（新图当然也会包含 v 中的顶点）。

接下来的清单 8.4 展示了对图 8.2 进行处理的过程。输出的是合并后的图中所

有顶点以及它们的连接关系。

清单 8.4 在哲学家关系图上进行 mergeGraphs 操作

```
val philosophers = Graph(
sc.makeRDD(Seq(
    (1L, "Aristotle"),(2L,"Plato"),(3L,"Socrates"),(4L,"male"))),
sc.makeRDD(Seq(
    Edge(2L,1L,"Influences"),
    Edge(3L,2L,"Influences"),
    Edge(3L,4L,"hasGender"))))
val rdfGraph = Graph(
    sc.makeRDD(Seq(
        (1L,"wordnet_philosophers"),(2L,"Aristotle"),
        (3L,"Plato"),(4L,"Socrates"))),
    sc.makeRDD(Seq(
        Edge(2L,1L,"rdf:type"),
        Edge(3L,1L,"rdf:type"),
        Edge(4L,1L,"rdf:type"))))
val combined = mergeGraphs(philosophers, rdfGraph)

combined.triplets.foreach(
      t => println(s"${t.srcAttr} --- ${t.attr} ---> ${t.dstAttr}"))
```

输出如下：

```
Socrates --- Influences ---> Plato
Plato --- Influences ---> Aristotle
Socrates --- hasGender ---> male
Plato --- rdf:type ---> wordnet_philosophers
Aristotle --- rdf:type ---> wordnet_philosophers
Socrates --- rdf:type ---> wordnet_philosophers
```

Scala 小贴士：Scala 提供了 "${myVar}" 类型的语法糖，以方便格式化输出长字符串的连接。

8.2 读取RDF图文件

尽管 GraphX 具有表示属性图的能力，但是仍有不少图数据是只使用三元组的形式来表示的。一个三元组（triple）包含了主语（subject）、谓语（predicate）和宾语（object），如图 8.3 所示。在 GraphX 中，主语和宾语都是指顶点的字符串类型的属性，谓语是指边的属性。存储三元组的标准文件格式多种多样，例如 RDF(Resource Description Framework) 和 N3(Notation 3)。

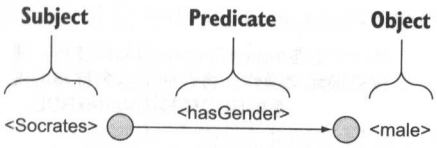


图 8.3 RDF 三元组示例。

一份众所周知的 RDF 数据来自于德国马克斯普朗克研究所的 YAGO3（Yet Another Great Ontology）。它来自 Wikipedia、WordNet、DBPedia、GeoNames 和其他数据源，尝试构建一个通用的本体系统。YAGO3 的全量数据包含了超过 50 亿个三元组关系，总大小为 90GB。YAGO3 的用途广泛，包括用于自然语言的处理和文本概念间关系的查找等。8.3 节会介绍它的一个简单用例，查找 Wikipedia 中缺失或前后不一致的数据。这是可行的，因为 YAGO3 的数据本来就大量来源于 Wikipedia。

8.2.1 顶点匹配以及图构建

在本小节中，我们会展示读取 RDF 文件的基本方法。下载 YAGO3 后会得到数十个 .tsv 文件（使用 Tab 作为分隔符）。以下是 yagoFacts.tsv 的示例内容：

```
<id_10silts_1sv_1rii7g7> <Plato> <influences> <Aristotle>
<id_10silts_p3m_zkjp59> <Plato> <hasGender> <male>
```

第一列是 ID，我们不需要处理。第二列是主语，第三列是谓语，第四列则是宾语。请注意，这里没有顶点 ID 的数据。所以读取 RDF 文件时，主要任务是根据顶点名称自动创建顶点的 ID。

接下来，我们首先创建顶点的字典 v。然后通过一系列 `join()` 和 `map()` 操作，根据字典 v 完成顶点名称到顶点 ID 的映射。在最后一个 `map()` 操作中，我们还会

为每一个三元组创建一个边的实例，参见清单 8.5。

清单 8.5 读取 Tab 分割的 RDF 文件

```
def readRdf(sc:org.apache.spark.SparkContext, filename:String) = {
  val r = sc.textFile(filename).map(_.split("\t"))
  val v = r.map(_(1)).union(r.map(_(3))).distinct.zipWithIndex
  Graph(v.map(_.swap)
        r.map(x => (x(1),(x(2),x(3))))
         .join(v)
         .map(x => (x._2._1._2,(x._2._2,x._2._1._1)))
         .join(v)
         .map(x => new Edge(x._2._1._1, x._2._2, x._2._1._2)))
}
```

使用与mergeGraphs函数类似的方法来创建一个包含不重复顶点的字典。

另一个借鉴mergeGraphs的编码技巧：通过交换元素操作，将存储顶点名称与ID关系的RDD转换成VertexRDD。

由于上述代码的匿名 Scala 元组嵌套使用超过了 3 层，难以直观地理解，我们在图 8.4 中详细说明了处理的具体过程。

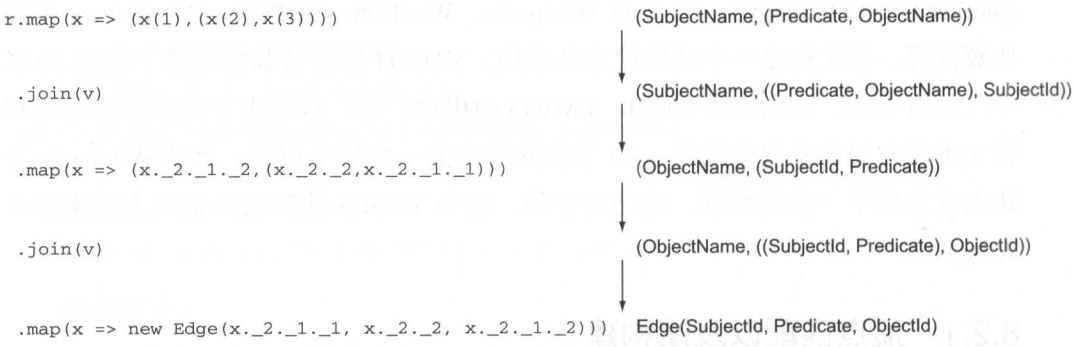


图 8.4 清单 8.5 中的数据流细节展示。首先将主语顶点（即起始点）作为顶点的 ID，然后再对宾语顶点（即目标点）执行类似的操作。最后使用这两个顶点 ID 和原始的边属性来构建新的边 Edge()。

测试 `readRdf()` 功能时，下载 YAGO3 的 yagoTypes.tsv 作为示例数据：

```
wget http://resources.mpi-inf.mpg.de/yago-naga/yago/download/
    ➥ yago/yagoTypes.tsv.7z
```

解压缩 .7z 格式的文件，具体的操作依赖于系统的不同，对于 CentOS（附录 A 中推荐的 Cloudera QuickStart VM 的操作系统），可使用以下操作：

```
wget http://packages.sw.be/rpmforge-release/
➥ rpmforge-release-0.5.2-2.el6.rf.i686.rpm
```

```
sudo rpm -ivh rpmforge-release-0.5.2-2.el6.rf.i686.rpm
sudo yum install p7zip
7za e yagoTypes.tsv.7z
```

现在我们可以看到，yagoTypes.tsv 的文件大小是 1.5GB，当在本地运行 Spark REPL 环境时，默认分配的 JVM 大小只有 500MB。因此，出于性能的考虑（以及可运行的配置考虑），你需要以 --driver-memory 的方式来启动 Spark REPL。同时，也请使用 --driver-cores 命令来启用机器上的所有核（对于 Intel i5 处理器，该数值可设为 4，但若为 Intel i7 处理器，该数值可增加到 8）。

```
./spark-shell --driver-memory 2g --driver-cores 4
```

一旦进入了 Spark REPL 环境，使用以下代码启动程序：

```
val gTypes = readRdf(sc, "yagoTypes.tsv")
```

然后，你可以下载另外一个 YAGO3 文件：

```
wget http://resources.mpi-inf.mpg.de/yago-naga/yago/download/
    ➥ yago/yagoSimpleTaxonomy.tsv.7z
```

现在就可以在 Spark REPL 中使用 mergeGraphs() 来读入这个新文件，并将它与第一个文件的数据进行合并：

```
val gSimpleTaxonomy = readRdf(sc, "yagoSimpleTaxonomy.tsv")
val gMerged = mergeGraphs(gTypes, gSimpleTaxonomy)
```

最后，下载 yagoFacts 文件：

```
wget http://resources.mpi-inf.mpg.de/yago-naga/yago/download/
    ➥ yago/yagoFacts.tsv.7z
```

8.2.2 使用 IndexedRDD 和 RDD HashMap 来提升性能

你会注意到使用上述方法进行较大的 YAGO3 文件处理时性能并不高。使用 IndexedRDD 可以提升约 30% 的性能，它是由 AMPlab 开发的库，可以从 spark-packages.org 获取。

在 Spark 1.6 中，IndexedRDD 并不属于 Apache 官方的版本。Ankur Dave，GraphX 的最初主要贡献者之一，在 2014 年就进行了 IndexedRDD 的开发（大概在

还是 Spark 1.0 版本的时候），用于提升 GraphX 的性能。Jira 的 SPARK-2365，申请将 IndexedRDD 合并到官方版本，但现在仍未被接受。

与普通的 RDD 相比，IndexedRDD 主要有两方面的特性。一是，正如它的名字所描述的，RDD 内的元素会被加上索引，并和 Java 等其他语言的 HashMap 效果相似。二是，它是可变的。正是这种可变性，使得 IndexedRDD 更适合被并入 GraphX 中。因为目前哪怕只是做小小的改动，也需要为此重新创建一个全新的全图，效率不高。索引的特性使得 IndexedRDD 更容易应用到 Spark SQL 的模块中。但合并 IndexedRDD 到 Spark 并不是一件简单的事情，需要很多测试，因为它改变了 Spark RDD 不可变的设计初衷。而 Spark 的 RDD 血统关系（lineages）管理、懒实现（laziness）、断点（checkpoint）等关键特性，都是基于 RDD 的不可变性提出的。

我们在上文提到 IndexedRDD 和 HashMap 效果相似，其实这种说法并不完全正确。事实上，IndexedRDD 更胜一筹，如图 8.5 所示，因为底层的实现使用了字典树（一种特定的搜索树结构）而不是哈希表。每个 worker 节点对 RDD 的全部数据进行处理时，这种线性扫描的方式适用于 `map()` 操作，但并不适用于 `join()` 类型的操作。

清单 8.5 中的代码片段使用了两个 `join()` 操作，因此我们可以使用 IndexedRDD 来提升性能。改进后的版本如清单 8.6 所示。相比普通 RDD 的 `join()` 操作，IndexedRDD 提供了作为替代的 `innerJoin()` 操作。它提供了第二个参数列表来输入另一个参数——即，执行映射操作的函数。不能直接使用这种映射来处理，因为我们不希望保留 `join()` 操作所依赖的 key（即上述的主语或宾语），但 IndexedRDD 会强制在结果中保留键值。因此，为了使 `innerJoin()` 和我们平常使用的 `join()` 操作效果类似，我们需要将函数 `(id, a, b) => (a, b)` 作为参数传递进去，因为这和 `join()` 产生的 key-value 键值对的形式一致。

你会注意到我们的代码构成了一个完整的单机程序，而不是适用于 Spark REPL 的代码片段。这是有原因的。我们需要将 IndexedRDD 以外部 Jar 包的形式进行引用，而在 Spark REPL 中用命令行的操作来引入带有依赖关系的外部 Jar 包的操作会相当麻烦。但使用依赖管理器来进行处理则更为简单，如 Apache Ivy（属于 SBT 的组件）或者 Maven，并且可以方便地生成组装好的 Jar 包（assembly Jar，也即 Fat Jar，带依赖关系的 Jar 包）。Java 程序员应该对 Maven 比较熟悉，这里我们选择了 Maven 而不是 SBT，因为 Maven 不需要额外的操作和配置。清单 8.7 所示的是对应的 pom.xml 文件，放在目录 /readrdf 下。清单 8.6 则对应于文件 ~/readrdf/src/main/scala /readrdf.scala。

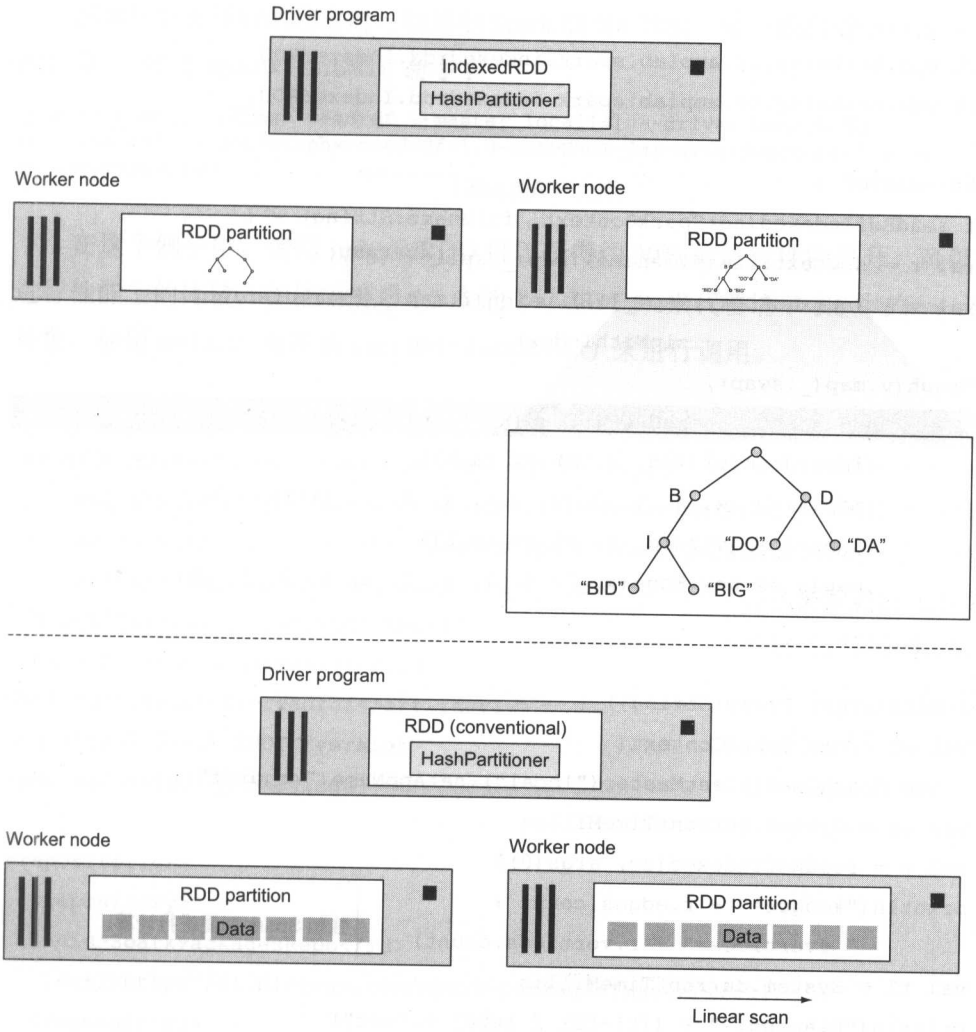


图 8.5 在 RDD 中通过 key 来找到指定的一条数据可分成两个步骤。第一步，Spark driver 节点使用 RDD 的 HashPartitioner 来定位到该数据所在的 worker，或者所在的分区。第二步，worker 节点在对应的 RDD 分区中找到执行的数据。IndexedRDD 和一般 RDD 的区别在于后面的这一步。在 IndexedRDD 中，worker 节点会使用字典树进行处理，而在一般的 RDD 中会使用线性扫描的方式。请注意，尽管字典树是图结构的形式，IndexedRDD 使用的是自定义的、高效的、非分布式的方式来实现字典树，而不是基于分布式的 GraphX 来实现。

清单 8.6 读取 Tab 分割的 RDF 文件，使用 IndexedRDD 提升性能

```
import org.apache.spark.graphx._
import org.apache.spark.{SparkContext, SparkConf}
```

```
import edu.berkeley.cs.amplab.spark.indexedrdd.IndexedRDD
import edu.berkeley.cs.amplab.spark.indexedrdd.IndexedRDD._

object readrdf {
  def readRdfIndexed(sc:SparkContext, filename:String) = {
    val r = sc.textFile(filename).map(_.split("\t"))
    val v = IndexedRDD(r.map(_(1)).union(r.map(_(3))).distinct
                       .zipWithIndex)
    Graph(v.map(_.swap),
          IndexedRDD(IndexedRDD(r.map(x => (x(1),(x(2),x(3)))))
           .innerJoin(v)((id, a, b) => (a,b))
           .map(x => (x._2._1._2,(x._2._2,x._2._1._1))))
           .innerJoin(v)((id, a, b) => (a,b))
           .map(x => new Edge(x._2._1._1, x._2._2, x._2._1._2)))
  }

  def main(args: Array[String]) {
    val sc = new SparkContext(
      new SparkConf().setMaster("local").setAppName("readrdf"))
    val t0 = System.currentTimeMillis
    val r = readRdfIndexed(sc, args(0))
    println("#edges=" + r.edges.count +
            " #vertices=" + r.vertices.count)
    val t1 = System.currentTimeMillis
    println("Elapsed: " + ((t1-t0) / 1000) + "sec")
    sc.stop
  }
}
```

强制 Spark 对图进行计算。

如果还没有安装 Maven（附录 A 的 Cloudera QuickStart VM 已预先安装了 Maven），请自行搜索在不同的操作系统中安装 Maven 的方法。一旦安装成功，使用以下指令进行编译：

```
cd ~/readrdf
mvn clean package
```

需要运行编译好的程序时，切换到 Spark 的 bin 目录（即一般启动 spark-shell 的目录），使用 spark-submit 命令：

```
./spark-submit --class readrdf --master local[4] --driver-memory 2g
➥ ~/readrdf/target/graphx-readrdf-1.0-SNAPSHOT-jar-with-dependencies.jar
➥ yagoTypes.tsv                    <-- 输入的RDF文件。
```

在这个例子中，使用 IndexedRDD 可以获得约 30% 的运行性能提升。可以继续进行性能上的比较，如使用清单 8.5 中的 readRDF() 来代替清单 8.6 中的 readrdf 对象，使用 main() 而不是 readRdfIndexed() 来进行调用。

清单 8.7 对应代码清单 8.6 的 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
     http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersio
  <groupId>com.manning</groupId>
<artifactId>graphx-readrdf</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<repositories>
  <repository>
    <id>SparkPackagesRepo</id>
    <url>http://dl.bintray.com/spark-packages/maven/</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.5.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
```

```
    <artifactId>spark-graphx_2.10</artifactId>
    <version>1.6.0</version>
  </dependency>
  <dependency>
    <groupId>amplab</groupId>
    <artifactId>spark-indexedrdd</artifactId>
    <version>0.3</version>
  </dependency>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.10.5</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.scala-tools</groupId>
      <artifactId>maven-scala-plugin</artifactId>
      <version>2.15.0</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.5.5</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
```

```
        </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

8.3 穷人（简化版）的图同构：找到Wikipedia缺失的信息

图的最有趣的应用之一就是找到图同构，即找到图中具有相似结构的两个部分，然后就可以使用一部分的信息来推断另一部分的信息。

例如，图 8.6 以类似 YAGO 的数据来进行阐述。

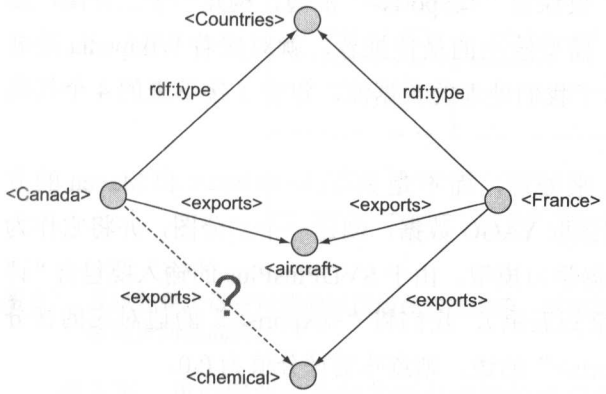


图 8.6 由 <France>、<Countries> 和 <aircraft> 组成的子图，与由 <Canada>、<Countries> 和 <aircraft> 组成的子图，是同构的。不仅是由于边的位置一致，它们边上的属性也是一致的。根据这幅图，我们能否推断加拿大（Canada）也会出口化学制品（chemicals）呢？

众所周知，YAGO 的数据是不完整的，因为它来自于 Wikipedia，而 Wikipedia 的数据依赖于志愿编辑者，他们在不同话题（甚至话题中不同的项目）花费时间的

多少，取决于话题的有趣程度。我们可以使用图同构来检测 YAGO 数据的不一致性。

首先我们会提供 YAGO 是如何使用 Wikipedia 数据的背景知识的。YAGO 数据中表示类别的边，如 rdf:type，来自于 Wikipedia 页面底部的类目标签。这些数据存放在 yagoTypes.tsv 文件中。而 YAGO 数据中表示事实的边，如“<exports>”和“<isLocatedIn>”则来自于 Wikipedia 页面右上角的消息盒。这些数据存放在 yagoFacts.tsv 文件中。

在一个 Wikipedia 页面中，类别标签和消息盒都是由志愿编辑者手工填写的，并具有高度不一致性。

图同构算法十分实用，但是本书不会讨论它的具体实现方法。相反，我们会用一种巧妙的处理方法。我们不会提供处理任意结构、任意大小的图的通用方法，而是仅处理“<exports>”的边以及找出可能缺失的“<exports>”边。我们利用到了“<exports>”的主语几乎肯定是国家的特性，从而不需要处理那些指向了“<Countries>”的边。我们同样会利用“<exports>”的宾语是产品类型顶点的特性来简化处理。在这样的前提条件下，我们仅处理“<exports>”的边，在大型图中进行匹配的操作会相当简单。

请注意，仅处理“<exports>”的边时，我们实际上是在寻找从国家顶点到商品顶点的二分图（二分图的更详细内容请查看第 3 章）。而基于二分图的结构，我们可以使用推荐系统进行处理，这里，我们会用到第 7 章介绍过的 SVDPlusPlus。

我们会在 YAGO 的完整图上，仅保留“<exports>”的边，构建一个二分图，然后在上面运行 SVDPlusPlus 算法。模型给出的最佳推荐，就对应着 Wikipedia 数据集中可能缺失的信息。图 8.7 展示了我们处理的数据流，包含了接下来的 4 个代码清单的内容。

我们的例子使用 Spark REPL 来处理，而不是 `spark-submit` 和 Maven 的方式。首先，如清单 8.8 所示，我们读取 YAGO 数据，创建一个二分图，并将它作为 SVDPlusPlus 的输入来训练一个机器学习模型。由于 SVDPlusPlus 的输入要包含“评分”值（例如，电影评论中的一星到五星），我们将“<exports>”的边对应的评分值设置为 1.0。那么不存在“<exports>”的边，则意味着评分值为 0.0。

清单 8.8 在 YAGO 的“<exports>”边集上训练 SVDPlusPlus 模型

```
val gf = readRdf(sc, "yagoFacts.tsv").subgraph(_.attr == "<exports>")
val e = gf.edges.map(e => Edge(e.srcId, e.dstId, 1.0))
val (gs,mean) = lib.SVDPlusPlus.run(e,
     new lib.SVDPlusPlus.Conf(2,10,0,5,0.007,0.007,0.005,0.015))
```

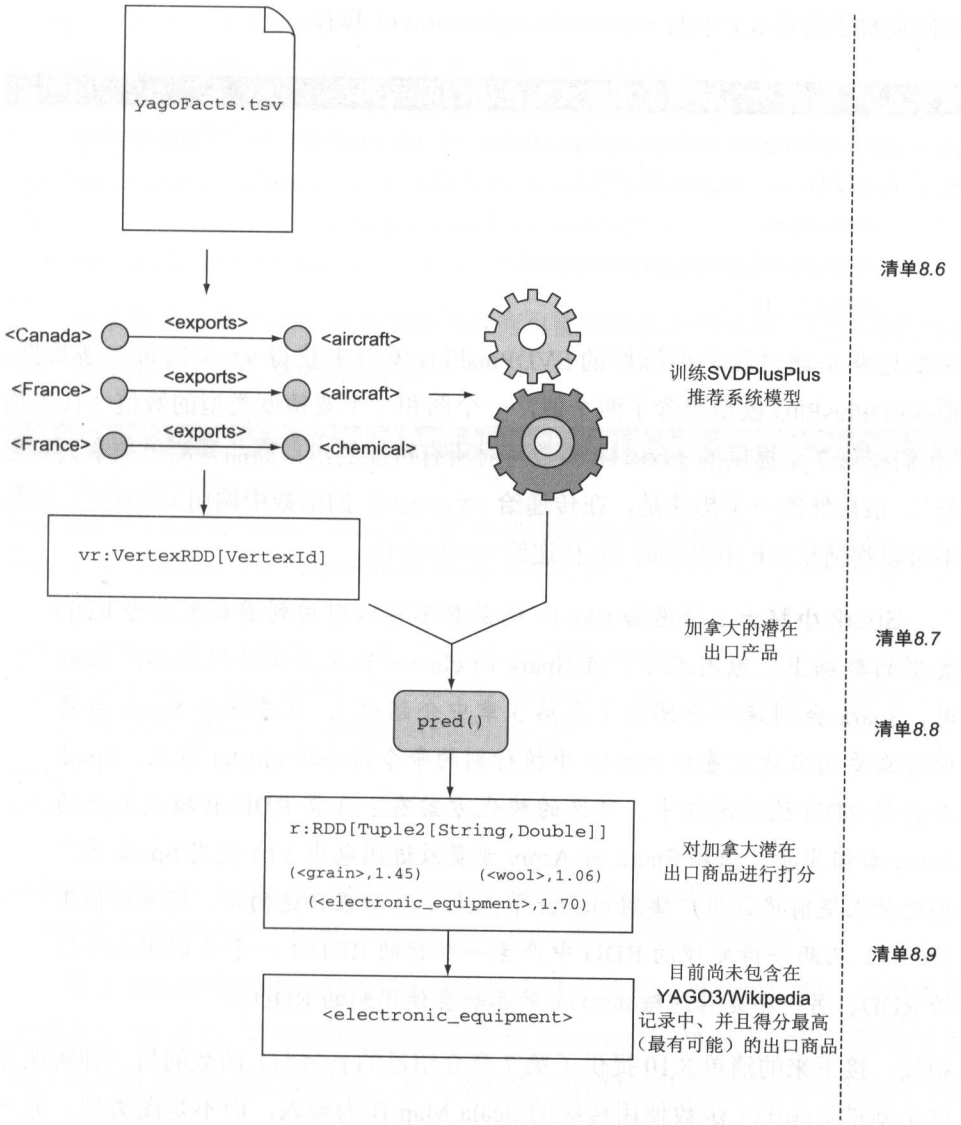


图 8.7 寻找加拿大最有可能的出口商品的数据流，这可能是 YAGO3 在处理 Wikipedia 数据时有所缺失而造成的。

接下来，我们需要一个包含所有可能的出口商品的列表，我们会对所有的国家的出口商品进行计算（不会考虑目前没有被任何国家出口过的商品的情况）。在我们的例子中，仅需要考虑加拿大的潜在的出口商品情况。清单 8.9 会生成一个命名为 vr 的 RDD，包含了加拿大所有可能的潜在出口商品（不包含已知的出口商品，会通过 subtractByKey() 去除掉，这里“subtract”指的是集合的操作）。请注意，

这段代码依赖于清单 8.1 中的 removeSingletons() 操作。

清单 8.9 计算 vr，缺失的加拿大潜在出口商品列表

```
val gc = removeSingletons(gf.subgraph(et => et.srcAttr == "<Canada>"))
val vr = e.map(x => (x.dstId,""))
          .distinct
          .subtractByKey(gc.vertices)
          .map(_._1)
```

主要思路是通过我们训练好的 SVDPlusPlus 模型来获得 vr 中的每一条数据。请记得 SVDPlusPlus 模型包含了两个值，一个图和一个双精度类型的数值（代表均值），我们在第 7 章提供的 pred() 函数可对所有可能的出口商品预测“加拿大对它的评分”。很自然的一个想法是，在传递给 vr.map() 的函数中调用 pred()。但是我们不可以将清单 8.6 中提供的 gs 传递给 vr.map()。

> **Spark 小贴士**：传递给 map() 的函数不可以引用到图类型或者 RDD 类型的数据上。原因在于，在 Spark 的 cluster 节点上初始化 map() 函数时，Spark 会创建一个闭包（在第 3 章中介绍过），这意味着 Spark 会将所有需要的本地变量和 map() 中执行到的命令传递给 cluster 节点。Spark 不会将 RDD 放进闭包中。可选的替代方案有：a) 将 RDD 转换成本地的 Array 数组变量，依赖 Spark 将 Array 变量放进闭包中；b) 使用 Spark 的广播变量来提前将数组广播到 cluster 节点上；c) 如果合适的话，还可以使用 join() 从两个待处理的 RDD 中产生一个新的 RDD（一个是调用 map() 的 RDD，另一个是传递给 map() 的函数会使用到的 RDD）。

相反，接下来的清单 8.10 提供了第 7 章介绍过的 pred() 函数的另一种实现方式。这个新的 pred() 函数使用传统的 Scala Map 作为输入，而不是图类型。另一个函数 vertexMap() 则会将 SVDPlusPlus 输出的图转换成我们需要的 Scala Map 类型的数据。

清单 8.10 map()——友好的 SVDPlusPlus 预测函数和工具

```
def pred(v:Map[VertexId, (Array[Double], Array[Double], Double, Double)],
         mean:Double, u:Long, i:Long) = {
  val user = v.getOrElse(u, (Array(0.0), Array(0.0), 0.0, 0.0))
  val item = v.getOrElse(i, (Array(0.0), Array(0.0), 0.0, 0.0))
```

```
  mean + user._3 + item._3 +
  item._1.zip(user._2).map(x => x._1*x._2).reduce(_ + _)
}

def vertexMap(g:Graph[(Array[Double], Array[Double],
                       Double, Double),Double]) =
  g.vertices.collect.map(v => v._1 -> v._2).toMap
```

接下来的清单 8.11 会计算加拿大最有可能的潜在出口商品。这有可能与事实相符，但却没有记录在 Wikipedia 上。

清单 8.11 查找没有记录在 Wikipedia 上的加拿大最有可能的出口商品

```
val vm = vertexMap(gs)
val cid = gf.vertices.filter(_._2 == "<Canada>").first._1
val r = vr.map(v => (v,pred(vm,mean,cid,v)))

val maxKey = r.max()(new Ordering[Tuple2[VertexId, Double]]() {
  override def compare(x: (VertexId, Double), y: (VertexId, Double)): Int =
      Ordering[Double].compare(x._2, y._2)
})._1
gf.vertices.filter(_._1 == maxKey).collect

res0: Array[(org.apache.spark.graphx.VertexId, String)] =
  Array((1721488,<wordnet_electronic_equipment_103278248>))
```

基于 YAGO 数据对 Wikipedia 缺失的边进行预测，最有可能的出口商品是“电子设备”。事实上，我们都知道加拿大公司 Research In Motion 会出口黑莓设备。但为了更严格地论证我们的结果，首先来看看 YAGO 提供的加拿大完整的出口商品列表：

```
grep "<exports>" yagoFacts.tsv | grep "<Canada>"
<id_1wrx1wu_dv6_1pgb7a4> <Canada> <exports> <wordnet_aluminum_114627820>
<id_1wrx1wu_dv6_j2l8e6>  <Canada> <exports> <wordnet_electricity_111449907>
<id_1wrx1wu_dv6_t6wmo1>  <Canada> <exports> <wordnet_lumber_114943580>
<id_1wrx1wu_dv6_jhowo0>  <Canada> <exports> <wordnet_natural_gas_114960090>
<id_1wrx1wu_dv6_s9bzqx>  <Canada> <exports> <wordnet_aircraft_102686568>
<id_1wrx1wu_dv6_12fzkgg> <Canada> <exports> <wordnet_plastic_114592610>
```

然后，根据 www.worldstopexports.com/canadas-top-exports/2502 的记载，位于前十的出口商品如下所示：

1. 石油——US$128,926,515,000（占总出口商品的 27.2%）
2. 车——$59,753,479,000（12.6%）
3. 机器、引擎、泵——$32,600,025,000（6.9%）
4. 宝石、贵重金属、硬币——$21,518,760,000（4.5%）
5. 电子设备——$13,639,592,000（2.9%）
6. 塑料——$13,192,128,000（2.8%）
7. 木材——$12,686,263,000（2.7%）
8. 航空器、航天器——$12,409,459,000（2.6%）
9. 铝合金——$8,865,363,000（1.9%）
10. 谷类食品——$8,774,059,000（1.8%）

YAGO 数据在铝合金、航空器、木材和塑料等条目上是正确的，但它没有记录电子设备，而电子设备的排名比以上条目的排名都要靠前。

上述过程仅做了单一条目的推荐。实际上，相比清单 8.11 仅查找 `maxKey` 的方式，我们应该为预测分数提供一个阈值，所有高于阈值的边都应该用于效果的检验。

也请注意，在使用 SVDPlusPlus 进行简化版的图同构检测时，一个主要的优点其实与图同构并不相关。正如第 7 章所提及的，SVDPlusPlus 使用了隐性变量进行处理，这意味着 SVDPlusPlus 也会推断出一些没有存在于图中的虚拟顶点。为了更加了解加拿大和电子设备的例子中隐性变量所起的作用，我们发现 YAGO 数据中出口电子设备商品的国家只有列支敦斯登（欧洲）。然后查看列支敦斯登的其他出口商品，发现只有“<hardware>”（硬件）一项。而硬件根本不属于加拿大的出口商品列表。因此 SVDPlusPlus 一定是形成了一个表示硬件与木材（lumber）相似的隐性变量，从而加拿大的出口商品会与列支敦斯登的相似。

8.4 全局聚类系数：连通性比较

在第 5 章中，我们使用了 GraphX 内建的三角形个数统计（Triangle Count）作

为衡量连通性的标准。另一种衡量方式则是全局聚类系数，它返回的值介于 0 与 1 之间，因而使用起来更方便，可用于不同大小的图之间连通性的比较。例如，在比较耶鲁毕业生的社交网络和哈佛毕业生的社交网络时，你可以直接通过全局聚类系数进行比较，尽管这两个网络包含的毕业生数目有差别。而与三角形个数统计等方法比较，全局聚类系数的一个缺点是需要耗费更多的计算资源。请注意不要将全局聚类系数与相关联的局部聚类系数混淆。

全局聚类系数的定义如下：

$$\frac{\#\ of\ closed\ triplets}{total\ \#\ of\ triplets\ (open\ or\ closed)}$$

三元组（triplet）指的是彼此间有两条或三条边相连的三个顶点组成的集合。如果有三条边，那它们就组成了一个三角形，我们称为封闭三元组（closed triplet）。如果只有两条边，我们称为开放三元组（open triplet）。对每个顶点都计算它所属的三元组个数。这意味着对一个三角形而言，我们一共可以得到三个封闭的三元组，因为每个顶点都对应一个与之相关的三元组。具体请见图 8.8 的说明。

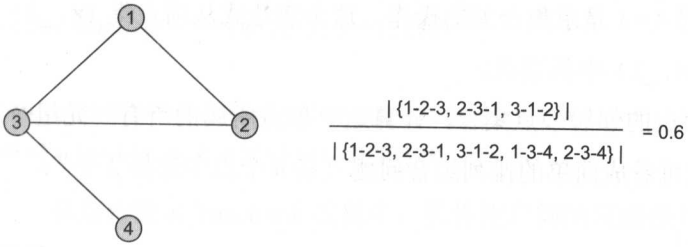


图 8.8 计算全局聚类系数的示例。上方的三角形构成了三个封闭三元组：一个与顶点 1 相关，一个与顶点 2 相关，一个与顶点 3 相关。开放三元组一共有两个，都与顶点 3 有关，即 1-3-4 和 2-3-4。

清单 8.12 定义了 `clusteringCoefficient` 函数，输入为 `Graph` 对象，输出是全局聚类系数。在 GraphX 的计算过程中，我们使用 `aggregateMessages()` 进行一次迭代，来允许每个顶点对它的所有邻居顶点列表进行计算。

清单 8.12 全局聚类系数

```
import scala.reflect.ClassTag
def clusteringCoefficient[VD:ClassTag,ED:ClassTag](g:Graph[VD,ED]) = {
  val numTriplets =
    g.aggregateMessages[Set[VertexId]](
        et => { et.sendToSrc(Set(et.dstId));
                et.sendToDst(Set(et.srcId)) },
        (a,b) => a ++ b) // #A
     .map(x => {val s = (x._2 - x._1).size; s*(s-1) / 2})
     .reduce(_ + _)

  if (numTriplets == 0) 0.0 else
    g.triangleCount.vertices.map(_._2).reduce(_ + _) /
      numTriplets.toFloat
}
```

aggregateMessages函数生成了一个VertexRDD，每个顶点都包含与它相连的其他顶点的ID组成的集合。

这里我们使用 Scala `Set` 进行处理，它和其他语言的集合操作相似（一般会自动去重，在我们的情况中则有自动过滤重复边的效果）。我们传递给 `aggregateMessages()` 作为 `mergeMsg` 参数的函数，使用的是集合的 `++` 操作，用于进行集合的合并。过滤操作的最后一步，我们去除了图中的自环（边的起始点和结束点是同一顶点），因为不希望把这种情况也看作是三元组。我们使用表达式 `x._2-x._1` 来完成上述目的，这里的减号（`-`）是求集合差的操作。这个表达式从顶点集（`x._2`）中去除了也包含在顶点集（`x._1`）中的顶点。

一旦我们获得了每个顶点的邻居顶点集合，计算这个顶点所属的所有三元组个数（包括封闭和开放的）就可看成简单的排列组合问题“从 n 个点中选择 2 个”：

$$\binom{n}{2}=\frac{n!}{2!(n-2)!}=\frac{n(n-1)}{2}$$

在分子的计算上，我们使用 GraphX 内建的三角形个数统计方法（Triangle Count）。它会为每个顶点计算顶点所属的三角形个数，然后我们使用 `reduce(_+_)` 将它们加在一起，所得的最终结果为实际三角形个数的 3 倍——这刚好是全局聚类系数公式的分子部分。

作为例子，我们从斯坦福网络分析项目中下载匿名的 Facebook 数据：

```
cd ~/Downloads
wget https://snap.stanford.edu/data/facebook.tar.gz
tar xzvf facebook.tar.gz
```

在众多的数据集中，我们选择第一个，即 set 0（参考清单 8.13）。文件 0.edges

可直接使用 GraphX 的 GraphLoader 进行读取。在审阅文件 0.featnames 时，我们可以看到第 77 个特征表示性别，所以我们在使用 sc.textFile 读取文件 0.feat 时将这个特征转换为布尔变量。

清单 8.13 全局聚类系数的使用例子

```
import org.apache.spark.graphx._
val g = GraphLoader.edgeListFile(sc, System.getProperty("user.home") +
  "/Downloads/facebook/0.edges")
val feat = sc.textFile(System.getProperty("user.home") +
  "/Downloads/facebook/0.feat").map(x =>
  (x.split(" ")(0).toLong, x.split(" ")(78).toInt == 1))
val g2 = g.outerJoinVertices(feat)((vid,vd,u) => u.get)

clusteringCoefficient(g2)
res1: Double = 0.8517387509346008

clusteringCoefficient(g2.subgraph(_ => true, (vid,vd) => vd))
res2: Double = 0.8881188035011292

clusteringCoefficient(g2.subgraph(_ => true, (vid,vd) => !vd))
res3: Double = 0.8304622173309326
```

在这份匿名 Facebook 数据中，男性用户间的连通性与女性用户间的连通性具有差异。

8.5 小结

- 一部分图处理系统会使用到的算法，在 GraphX 中没有现成的版本。
- GraphX 中的 subgraph() 函数会保留孤立顶点，我们提供了清除这些孤立顶点的代码。
- 合并图操作相当有用，如对 YAGO3 而言，一个图会被拆分为多个更小的子图。
- RDF 是标准的文件格式，我们提供了解析和匹配顶点名字的代码，用于构建 GraphX 所需的图。

- 图同构是查找关系和从 YAGO3 类型的图中获得值的有效方法。推荐系统可用于一些简单的图同构识别问题中，简单高效。
- 全局聚类系数是连通性的衡量指标，类似 GraphX 内建的三角形个数统计方法的作用。但它返回的是介于 0 与 1 之间的标准值，可方便地用于不同大小的图之间连通性的比较。

9 性能和监控

本章要点

- Spark 应用监控
- 性能相关的配置项
- 应用性能调优
- 使用图分区来增强大规模图处理

到目前为止我们演示的大部分实例都是小规模的应用，可以单机运行而不需要大规模的计算资源。但是，我们选择 Apache Spark 的关键因素是利用 Spark 的分布式处理能力。在多台机器上分布数据以及处理数据是 Spark 的核心能力，即我们所说的大规模的数据集处理。

一旦有了大量的机器资源并且都安装了 Spark，要让 Spark 应用在一个大数据集上运行可能还需要一些规划、配置和排查问题的能力。在本章，我们会带领大家通过一些必要的步骤来成功地运行 Spark 应用，以及讨论如果 Spark 应用出现问题应该如何排查。在这个过程中我们会让你更深入地理解 Spark 的数据处理模型，以及了解许多 Spark 配置应该如何设置。

9.1 监控Spark应用

如果你已经有了数据源，建立了基于图算法的数据挖掘模型，写代码实现并且测试通过，现在要在实际的大数据上运行你的图算法。你如何确保应用运行得尽可能快，或者性能达到最优？

Spark 有很多不同的 API 和配置设置，可以用来达到最佳性能。在花费大量时间试图优化 Spark 和应用之前，完全有必要先清楚理解 Spark 是如何运行分布式计算的。

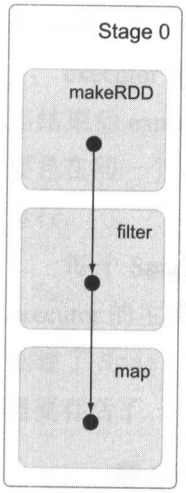
监控是理解 Spark 执行用户应用程序的必要方式，也是排查问题的关键工具。Spark 提供了许多用户接口来追踪应用运行时的状况。首先了解一些核心概念，然后再深入 Spark 提供的各种监控工具。

9.1.1 Spark 如何运行应用

当一个 Spark driver 程序执行了 `count`、`reduce` 动作或者把图 RDD 的输出写磁盘，会发生什么事情呢？这一小节我们来详细了解 Spark 如何执行一些简单的作业，以及如何用 Spark 监控工具发掘 Spark 执行作业的信息，随后再用这些知识来进一步了解 Spark 在集群中执行大规模作业时做了什么事情。这一小节介绍的内容在诊断和解决运行的作业问题时选择正确的优化方案是非常有帮助的。

如图 9.1 所示，一个作业从文本文件读取数据，过滤出包含特殊字符的行，然后把每行数据全部改为小写。在这个过程中，创建了三个 RDD，然后在最后一个 RDD 上调用 action 函数（`collect`）。Spark driver 会分析 RDD 链，这个 RDD 链要求有输出并且要生成一个会在集群 worker 节点上执行的作业。

作业由一个或多个 stage 组成，也是一个在数据集上将要执行的操作的集合。之前提到过，RDD 中的数据被切分为多个分区，基于这些分区数据，在集群多个 worker 节点上执行（或应用）一些操作。而创建的 stage 包含一系列操作，这些操作会在每个分区数据集上执行不需要依赖其他分区的数据。这个示例中的 `filter` 和 `map` 会划分在一个 stage 里，因为初始化的 RDD 数据中的每条数据都是互相独立的，没有依赖关系。




```
val rdd1 = sc.textFile("LICENSE")

val rdd2 = rdd1.filter(_ contains "Apache")

val rdd3 = rdd2.map(_.toLowerCase)
rdd3.collect // action that initiates job
```

图 9.1 一个调用了 filter 和 map 操作的作业只需要一个 stage。这两个操作执行在每条数据相互独立的 RDD 数据集上。

还有些作业，其中的数据需要在多个分区上洗牌，最终得到预期的数据结果。比如 groupByKey，键相同的元素从多个分区上分发出来，即通过键对数据分组，需要将所有的数据中键相同的数据放到同一个分区上。Spark 会把这个洗牌前和洗牌后的两个处理过程分成两个不同的 stage 阶段。

定义：当处理过程像上面一样必须被分割成多个 stage 阶段时，数据流分割的点就被称为洗牌边界，引起洗牌的转换被称为宽转换（相反就是窄转换）。

图 9.2 所示的代码如清单 9.1 所示。

清单 9.1 要求多个 stage 的转换

```
val rdd = sc.makeRDD(1 to 10000)
rdd
  .filter(_ % 4 == 0)
  .map(Math.sqrt(_))
  .map(el => (el.toInt,el))
  .groupByKey
  .collect
```

“1 to 10000”：用Scala的方式生成一个指定范围的数字集合。

“窄”转换仅仅在一个分区上做数据转换操作。

groupByKey：宽转换，生成的结果数据是通过洗牌的方式从多个分区上汇集而来的。

groupByKey 强制 Spark 创建两个 stage 阶段，对于更复杂的代码可以有大量的 stage，以便产生一个单一的结果。

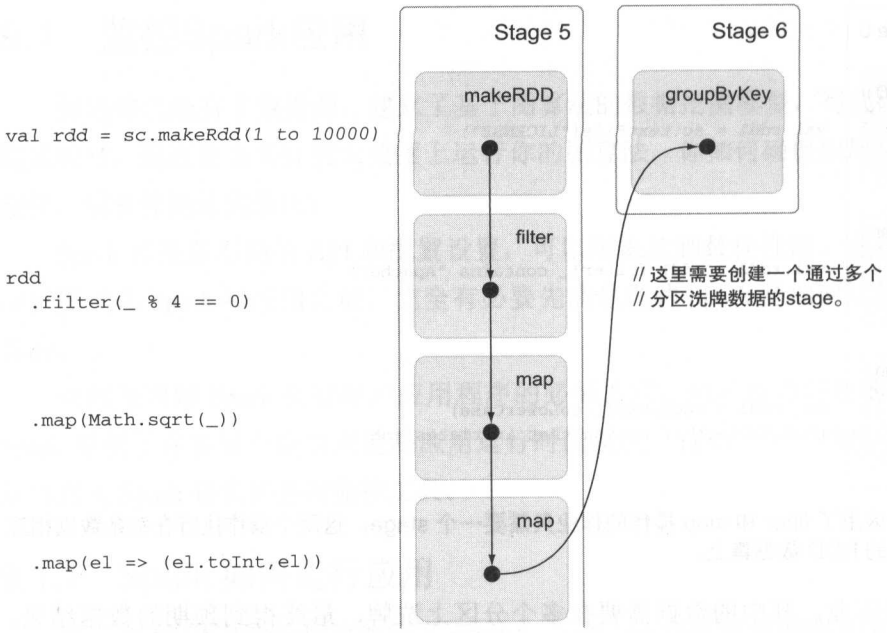


图 9.2 需要跨分区检查数据，如 groupByKey 操作，在跨 stage 边界的地方导致洗牌。

如图 9.1 和图 9.2 所示，stage 是一个或更多转换操作的集合，或者用官方名字：task。

定义：task 是 Spark 中最小的工作执行单元。一个 task 表示：被调度的处理任务在 Spark 集群的 worker 节点上执行。

想一下清单 9.1 所示的 Spark 作业示例，假定数据有 4 个分区。当第一个 stage 执行时，数据处理会被分布在 4 个 worker 节点上的 task 各自执行。当一个 worker 节点执行 task 任务时，它会在这个子数据集上应用 stage 操作（这里是 filter 和 map 操作）。task 的结果输出提供给下一个 stage 作为输入，即这里的 `groupByKey`。然后进行洗牌操作：把数据写到磁盘，然后通过网络读取其他分区的数据，把键相同的数据放到同一个 worker 节点上做进一步处理。

因为对数据进行处理要局限于一个 stage 阶段（不需要依赖任何其他分区中的数据），所以 task 任务可以并行进行。

executors：任务实际执行的地方

前面我们了解了将要在 worker 节点上执行的 task 任务，下面我们再来学习一个

与 worker 如何执行 task 任务相关的概念。

executor 是在 worker 节点上启动的一个进程，运行在应用的生命周期中，即应用结束后 executor 也会跟着退出。executor 的职责是服务于一个指定的应用，所以要是在同一个 worker 节点上运行多个应用，那么对应的就有相同数目的 executor 在运行。

每个 Spark 应用启动后都会向集群管理器请求多个 executor（见图 9.3）。executor 的主要任务是接收和处理作业调度器发来的 task 任务，作业调度器运行在创建了 SparkContext 的 driver 程序中。当 Spark 应用完成后，executor 也就不再需要存活了。

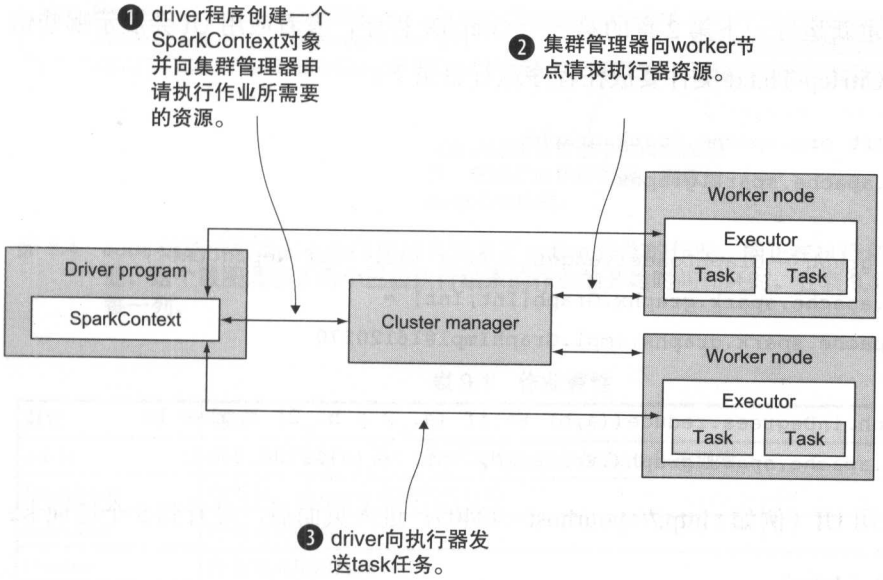


图 9.3 driver 程序创建了 SparkContext，然后通过集群管理器在 worker 节点上创建 executor。SparkContext 中有一个作业调度器（没有标示出来），用来向执行器分发要执行的 task 任务。

创建 SparkContext 时，会配置所需的资源（CPU 核数和内存）。SparkContext 向集群管理器请求创建足数的 executor 以提供所需的资源。在本章第 2 节会看到用于配置和优化集群资源的选项。

9.1.2 用 Spark 监控来了解你的应用的运行时信息

Spark 有不少监控工具，用于了解已经完成的和正在运行的作业。这一小节我

们会了解到应用 UI 的一些关键特性，这个应用 UI 是一个基于 Web 的监控应用，是诊断应用问题的重要工具。

应用 UI

一旦在应用中创建了 SparkContext，同时也就创建了 Web UI 来展示应用的各种信息。启动的 Web server 默认占用 4040 端口，要是 4040 端口被占用了，会依次累加端口号来尝试占用（4041、4042 依次递增）。

注意，spark-shell 会自动创建 SparkContext，所以当运行了 spark-shell，应用 UI 也就是可以访问的了。当你编译和构建自己的应用程序时（如第 3 章所示），创建了 SparkContext 后，应用 UI 才可用。

让我们重新运行一下第 2 章的第一个 GraphX 程序，看看应用 UI 会展示哪些信息，注意，CitHep-Th.txt 文件要放在程序运行目录下。

```
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._

scala> val graph = GraphLoader.edgeListFile (sc, "Cit-HepTh.txt")
graph: org.apache.spark.graphx.Graph[Int,Int] =
     org.apache.spark.graphx.impl.GraphImpl@16120270

scala> graph.inDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
res0: (org.apache.spark.graphx.VertexId, Int) = (9711200,2414)
```

打开应用 UI（例如 : http://<yourhost>:4040），进入页面后，可看到 5 个选项卡：

- 作业（Jobs）
- 阶段（Stages）
- 存储（Storage）
- 环境（Environment）
- 执行器（Executors）

在这一节我们来看看“作业”、“阶段”、“环境”选项卡中都有哪些信息，包括非常有用的事件时间线和 DAG 可视化图形工具。（“存储”和“执行器”选项卡放在后面的章节介绍）。监控页默认显示的是 Jobs 选项卡，如图 9.4 所示。

Jobs 选项卡展示了在应用生命周期中运行的作业列表，如表 9.1 所示，列出了作业许多有用的属性。

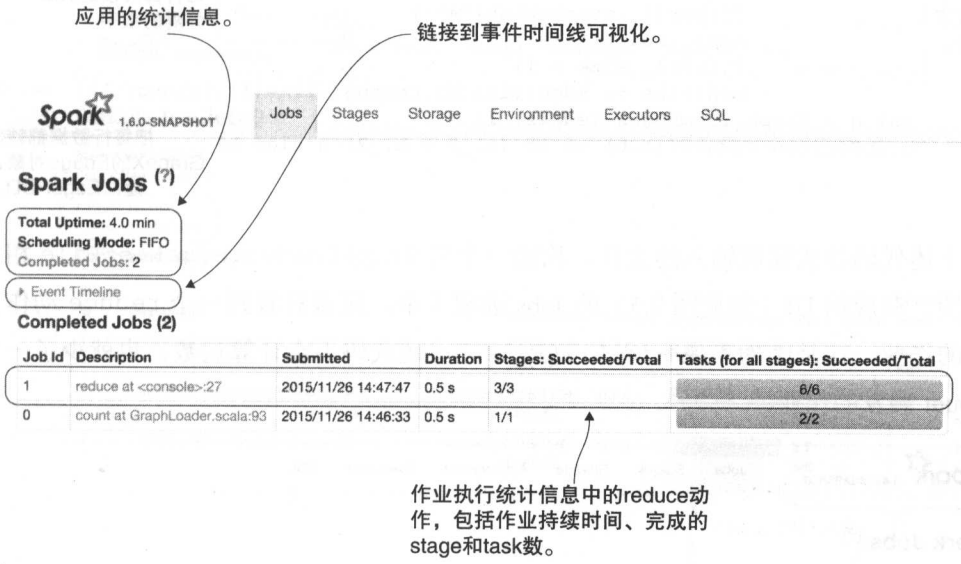


图 9.4 Jobs 选项卡列出了在应用中所有触发了 action 操作的作业，即正在执行和已经执行的作业。这个截图展示了两个已经执行结束的作业的开始时间和耗时。在表 9.1 中会详细介绍每一列。

表 9.1 作业属性

属性	描述
Job Id	第一个作业的 ID 是 0，然后每个连续作业的 ID 依次递增
Description	触发执行的 action 和源代码行号
Submitted	作业提交时间
Duration	任务完成所用时间
Stages	stage 总数和成功完成的 stage 数，两个数相同表示作业成功结束；如果是一个运行中的作业，会看到这两个数字逐渐接近直到相同
Tasks	每个 stage 都由多个 task 任务组成，这一列用图形化来表示所有 stage 中的任务总数，以及已经完成多少个任务

如图 9.4 所示，`reduce` 动作触发的作业共耗时 0.5 秒，包含了 3 个 stage，共 6 个 task 任务。但是，ID 为 0 的作业是什么，它从哪里来的？其实它来自方法 `GraphLoader.fromEdgeFile` 的实现，里面包含了一个作业。我们来重启一下 `spark-shell`，自己通过使用 Cit-HepTh.txt 文件的数据构建 `EdgeRDD`，再传给函数 `Graph.fromEdges`：

```
import org.apache.spark.graphx._
val edgelist = sc.textFile("Cit-HepTh.txt")
val edges = edgelist
            .filter(!_.startsWith("#"))
            .map(_.split("\\s"))
            .filter(_.size > 1)
            .map(line => Edge(line(0).toLong, line(1).toLong, 1))
val g = Graph.fromEdges (edges, 1)
g.inDegrees.reduce((a,b) => if (a._2 > b._2) a else b)
```

忽略以#开头的注释行。

每行都应该是Tab分隔的数字。

忽略少于两个数字的行。

把每行数据都转成GraphX的Edage对象，返回EdgeRDD。

上述代码显式读取输入的文件，构造一个与 GraphLoader.edgeListFile 相同的图。在应用 UI（参见图 9.5）的 Jobs 选项卡中，应该只看到一个 reduce 动作触发的作业，这是因为不像 edgeListFile，上述代码不会计算边数，也就少了一个 count 触发的作业。

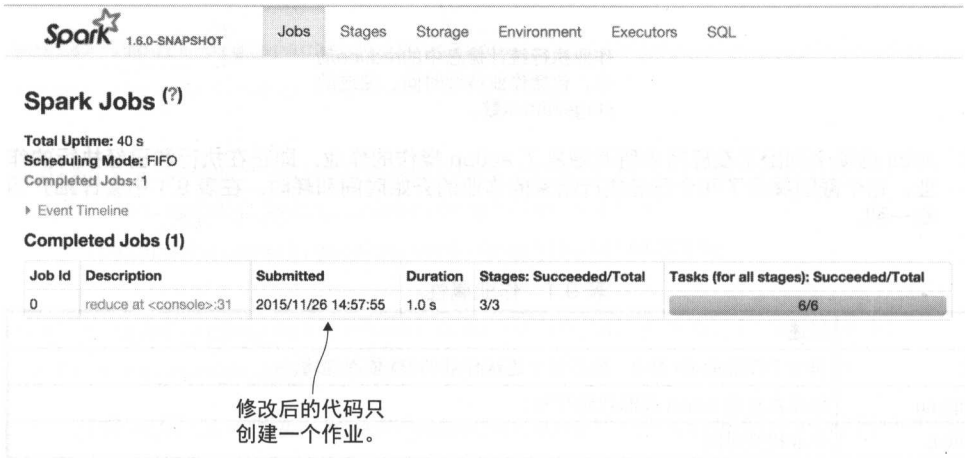


图 9.5 Jobs 选项卡现在只显示一个作业。

正如你所见，Spark UI 在追查应用运行时的状态时非常好用。

回到之前的有两个作业的例子，来看一下作业创建的 stage 中都有哪些信息。单击 Stages 选项卡，可看到所有的应用创建的所有 stage。说实话，要是应用中有大量的作业正在执行，那么 Stages 选项卡看起来会很乱。一般在调试或者调优的时候才会关心每个作业、每个 stage 以及组成作业的 task 任务的运行状态。回到 Jobs 选项卡，单击 reduce at <console>:x 链接，UI 打开一个如图 9.6 所示的作业的 stage 视图。

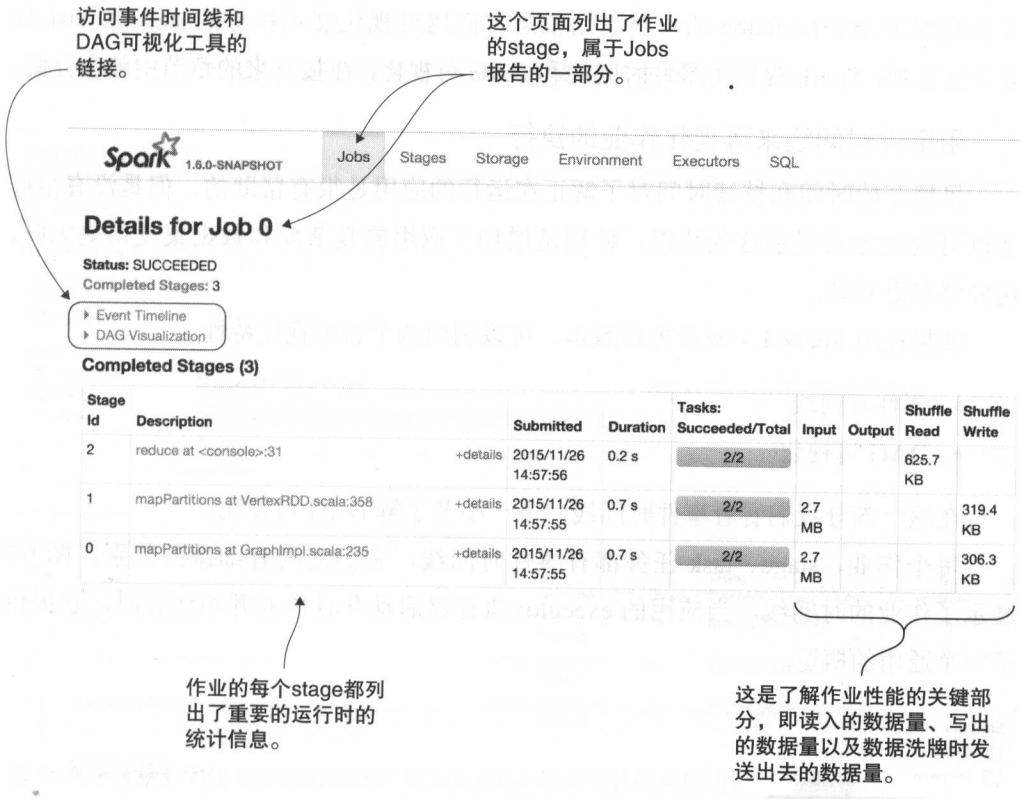


图 9.6 reduce 作业的 stage 列表。注意 Jobs 选项卡仍然是高亮的。

“Details for Job x”页面列出了选中作业的所有 stage，stage 的属性见表 9.2。

表 9.2 stage 属性列表

属性	描述
Stage Id	第一个 stage 的 ID 是 0，随之连续的 stage ID 依次递增
Description	由每个触发作业运行的 action 作为 stage 名（如 reduce、count），或者由生成的 shuffle 洗牌边界的转换函数名作为 stage 名
Submitted	调度器调度处理 stage 的开始时间
Duration	处理 stage 的耗时
Tasks	这一列以图形化表示 stage 的总任务数，以及有多少已经完成
Input	stage 中的所有 task 所读的总数据量
Output	stage 中的所有 task 所写的总数据量
Shuffle Read	DAG 中从前一个 stage 中读取的洗牌数据总量
Shuffle Write	传入到随后的 stage 的洗牌数据总量

这个用户界面主要专注于单个 action 的指标。通过提交时间和持续时间这两列，

可以洞察多个平行 action 的一些有用信息。而用可视化展示执行的 stage 和 task 任务会更直观，Spark 提供了事件时间线和 DAG 可视化，在接下来的章节中就会介绍。

用事件时间线来可视化作业的执行

虽然开始时间和持续时间对了解正在运行的应用是很有帮助的，但是没有很好地以可视化方式展示这些数据。特别是增加了应用的复杂度和数据集变得更大时，这种弊端更明显。

如果使用 Spark 1.4 或者更新版本，可以利用两个新可视化特性：

- 事件时间线
- DAG 可视化

在这一部分我们看看事件时间线，下一小节了解 DAG 可视化。

每个作业、stage、task 任务都有事件时间线，它们之间有细微的差别。图 9.7 展示了作业的时间线，当应用的 executor 执行器启动并且作业开始运行时，它提供了一个通用的概览。

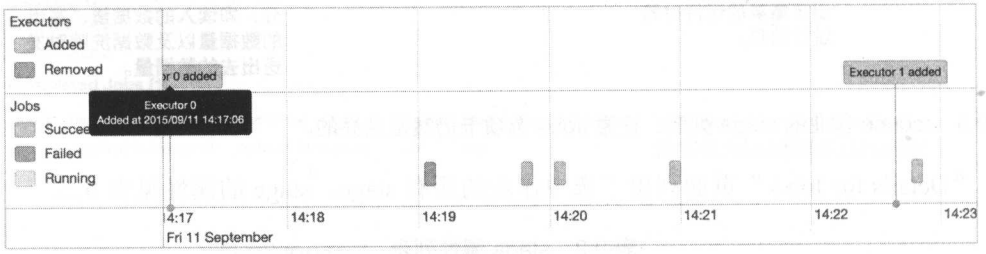


图 9.7 作业的时间线：4 个作业在第一个 executor 上运行；第二个 executor 执行器在 14:22 左右启动，启动后运行了一个作业。

单击作业，可看到运行的 stage 及其时间线。这个视图非常有用，可以了解 stage 之间在运行时的关系。这些 stage 是并行运行的，并且必须要等前面的 stage 结束后，下一个才能运行。

图 9.8 所示的是第一个 GraphX 应用程序实例的 stage 时间线。注意，加载并创建 Edge 和顶点 RDD，这两个操作是并行执行的，但必须等前面这两个操作执行完了才能开始执行 reduce 的 stage。

很明显，顶点 RDD 的 stage 执行得比边 RDD 要慢（时间线中的标签为：`GraphImpl.scala` 中的 `mapPartitions` 函数）。要优化和提升边 RDD 的 stage，

显然不会缩短总耗时。在这个例子中，优化最耗时的 `VertexRDD.scala` 中的 `mapPartitions` 函数 stage 以及 `reduce` 才能缩短整体耗时。

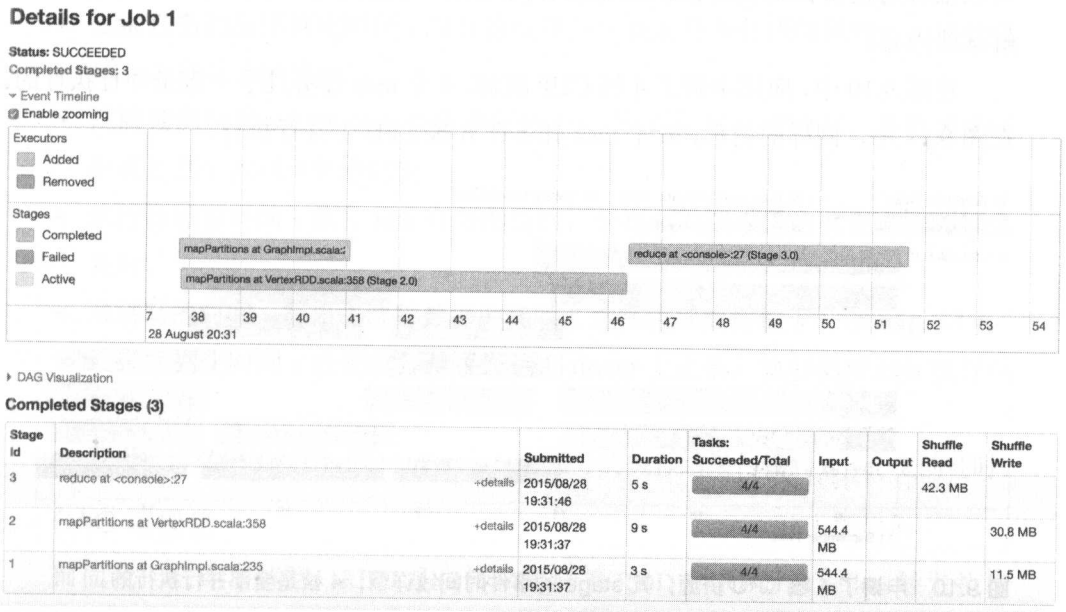


图 9.8 stage 时间线可以显示出哪些 Spark stage 阶段是并行运行的。

单击 stage 链接，打开 stage 详细页，可展示组成 stage 的全部 task 任务的度量指标，也包括 task 运行时的时间线。

Spark 应用的一个关键设计目标是，确保作业能获得最大限度的并行执行，这样可以充分利用 CPU 和内存资源。

图 9.9 和图 9.11 所示的是相同的 task 任务在不一样的时间线中的展现方式。在 task 执行时，应用根据预先配置的资源需求向集群申请不等核数的 CPU 资源。

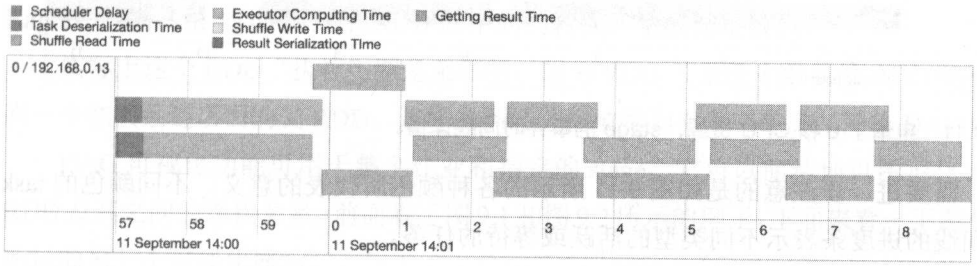


图 9.9 stage 的事件时间线详情展示了 task 任务执行的时间线，可以看到只有两个 task 任务曾并行执行过。

我们都期望有更多的 CPU 资源，有更多的 task 任务并行，这样应用运行得会更快。通过如图 9.9 所示的 task 任务的事件时间线，看一下实际的 task 任务并行度，可以很容易衡量在调优时修改配置或者代码所产生的变化。下面的章节会进一步了解这些内容。

在图 9.10 中，应用申请了 4 核 CUP 资源，4 个 task 任务几乎一直是并行执行的。在图 9.11 中，同样的情形，6 个 task 任务在 6 核 CPU 上并行执行。

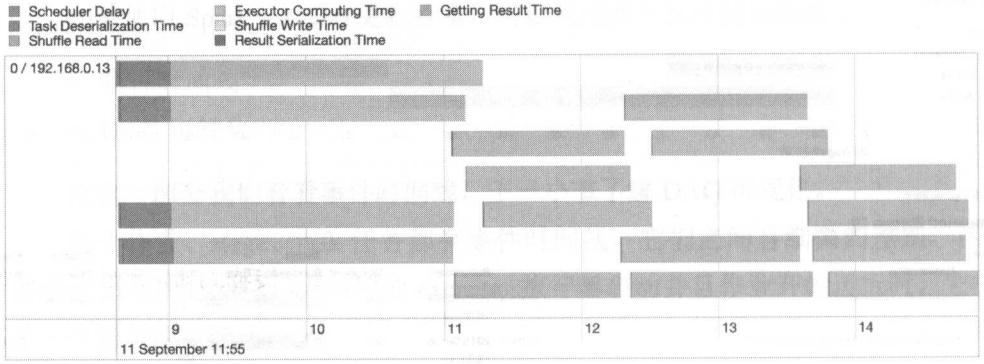


图 9.10 申请了 4 核 CPU 资源，其 stage 的事件时间线详情，4 核是全部并行执行的。

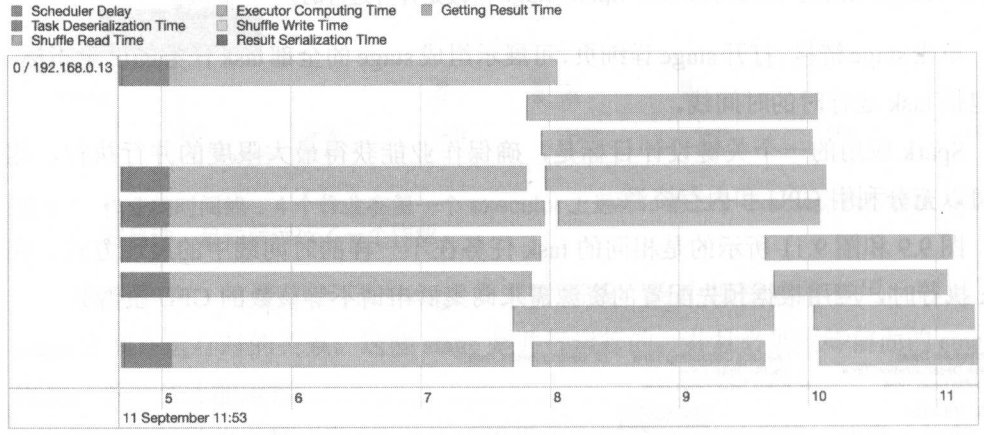


图 9.11 申请了 6 核 CPU 资源，stage 的事件时间线详情。

需要进一步注意的是如图 9.11 所示的各种颜色所代表的意义，不同颜色的 task 时间线的进度条表示不同类型的活跃或等待的任务。

- **调度器延迟**：在一个 task 任务被执行器执行之前，任务调度器需要把要执行

的 task 代码发送到执行器。调度器延迟时间就是 task 任务的网络传输耗时，即从任务调度器到 worker 上的执行器，等 task 任务执行结束再把结果传输回来。

- **task 任务的反序列化时间**：执行器反序列化 task 任务代码的耗时，反序列化后 task 任务才能执行。
- **洗牌读的时间**：如果 task 任务需要读上一个 stage 的洗牌数据，洗牌读的耗时就是其中的网络传输耗时。
- **执行器运算时间**：执行 task 任务的耗时，包括从 HDFS 或者其他数据源读的耗时。
- **洗牌写的时间**：写出去洗牌数据的耗时，这些洗牌数据被下一个 stage 读入。
- **结果序列化时间**：在把执行结果发回到 driver 上之前，执行器序列化执行结果的耗时。
- **获得执行结果的时间**：把 worker 上 task 任务的执行结果传输到 driver 上的耗时。

DAG 可视化

前面我们已经了解到，RDD 是由转换器转换 RDD 和 RDD 的 action 触发执行函数链接起来的。在一个 Spark 作业中，从上一个 RDD 读入数据，应用转换函数，生成结果数据，这个过程在 RDD 链中依次执行。我们写的 Spark 代码中一般都会包含许多前后链接的 RDD，这就组成了相当复杂的结构，要了解这些 RDD 在运行时如何展开会比较困难，DAG 可视化工具解决了这个问题。

在 RDD 关系链中，每个 RDD 都有一个或多个父 RDD，所以我们可以认为 RDD 是图的顶点，父子 RDD 之间的连接可以认为是边。计算结果在图中的方向是确定的，因为父 RDD 只能传递数据到子 RDD，数据不能以相反的方向流动。“根”RDD 从文件系统或其他数据源生成初始数据，例如数据库或流的数据源，如 Kafka，数据从根 RDD 经过子 RDD 的中间的转换，生成结果数据。

实际上这个 RDD“执行”图是无环的，因为 RDD 关系链中前面的 RDD 绝不会有一个在关系链后面的父 RDD。这样的顺序结构被称为有向无环图（简称 DAG）。

DAG 可视化功能可用于整个作业和指定的 stage，它可以简洁地以图形化表示 RDD 及其之间的连接关系。前面看了图 9.1 和图 9.2 所示的例子，下面来看一下图 9.12 所示的另一个可视化展示。

Spark 的作业可视化对查看作业的整体过程很有帮助，可以快速找到哪里有

RDD 缓存，缓存很关键。9.3 节有更多与缓存相关的内容。

单个 stage 的可视化对找出根 RDD 的数据源最为有用。如果数据源是基于文件的，它会标示出数据的文件路径。

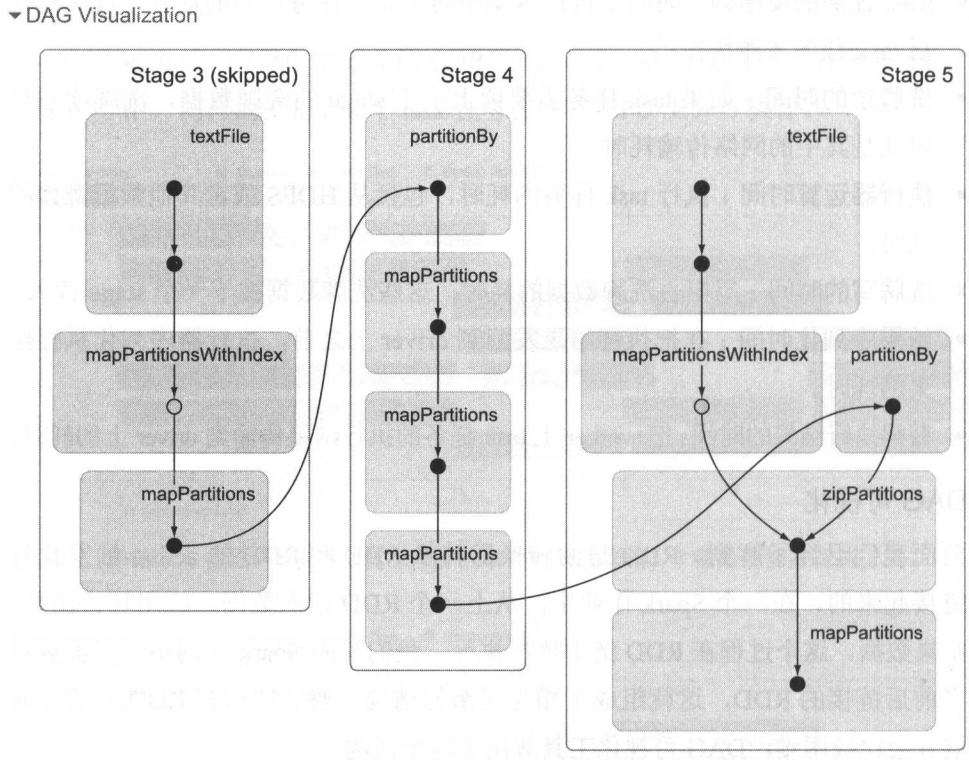
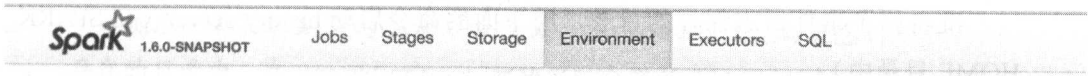


图 9.12 一个 Spark 作业的 DAG 可视化示例，stage 中有“跳过”标记的 RDD 表示 RDD 已经缓存了，可以直接从这个 RDD 的缓存读数据而不需要重新计算。

Environment 选项卡

Environment 选项卡展示了 driver 程序上设置的各种属性和 classpath 项，如图 9.13 所示。在诊断问题时，可以在这里确定属性对应的值。如果是运行 `spark-shell` 命令行或者用 `spark-submit` 提交一个本地作业（设置的 `master` 参数为 `local` 或 `local[n]`，即本地模式），在 driver 的 JVM 中还会创建一个 executor，那么在这个 Environment 选项卡上看到的是 driver 和 executor 的合集。

在非本地模式下，executor 不会和 driver 在同一个 JVM 中，很可能是在其他机器上。每个 worker 节点都会创建一个 Web UI 来提供它管理的 executor 信息，所以需要用 worker 节点上的 UI 来查看 executor 的环境变量信息。



Environment

Runtime Information

Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/jre
Java Version	1.8.0_40 (Oracle Corporation)
Scala Version	version 2.10.5

Spark Properties

Name	Value
spark.app.id	local-1448549843823
spark.app.name	Spark shell
spark.driver.host	192.168.0.13
spark.driver.port	54108
spark.executor.id	driver
spark.externalBlockStore.folderName	spark-76d6764e-57ce-4edf-b5ca-1f883b3aecaa
spark.fileserver.uri	http://192.168.0.13:54110
spark.jars	
spark.master	local[*]
spark.repl.class.uri	http://192.168.0.13:54107
spark.scheduler.mode	FIFO

图 9.13 Spark 的 Environment 选项卡（Spark UI 页面顶部菜单有访问链接）。

9.1.3 history server

应用 UI 的存活与应用创建的 SparkContext 息息相关，一旦应用创建的 SparkContext 停止（即调用了 SparkContext.stop），这个应用 UI 也就无法访问了。这意味着在应用程序结束后无法诊断问题，也不能重放历史来对比执行历史。这些问题可以通过 history server 解决。

history server 是长期运行的 Web 应用，它监控多个 Spark 应用创建的事件日志目录，并且重建应用 metric 度量信息及其相关的 UI。应用 history server UI 与前面看到的 SparkContext 停止之前的应用 UI 几乎一样。

要使用此功能，首先需要决定在哪里运行 history server。最简单的方法是在 driver 程序运行的机器上运行它。然后，需要为应用创建一个目录，用于将事件日志记录保存到这里。默认情况下是 /tmp/spark-events 目录，最好先创建一个自定义目录。

准备好了事件日志保存的目录，用下面的命令启动 history server（在 SPARK_HOME 目录中）：

```
./sbin/start-history-server.sh
```

这会启动一个监听 18080 端口的 Web server。

首先，Spark 应用启动时要把 `spark.events.enabled` 设置成 `true`，可以在 `spark-shell` 或 `spark-submit` 命令行后追加参数 `--conf "spark.eventLog.enabled=true"`：

```
spark-shell --conf "spark.eventLog.enabled=true"
```

加了这个参数，应用会保存事件到默认目录 /tmp/spark-events。当 Spark 应用结束后，记录的事件会保留在事件日志目录，不会清空。打开浏览器，输入 history server 的地址和端口（如：http://localhost:18080，这是 history server 本机），可以看到已经执行过的 Spark 应用。

如图 9.14 所示，首页列出了所有已经结束并记录了事件的应用，单击应用链接，会看到应用的 UI 界面，犹如应用的 `SparkContext` 仍然运行着。这么一来，就可以如前一节描述的那样来调查应用程序了。

Spark 1.5.0 **History Server**

Event log directory: file:/tmp/spark-events

Showing 1-2 of 2 1

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
local-1443006126350	Spark shell	2015/09/23 12:02:05	2015/09/23 12:07:23	5.3 min	robineast1	2015/09/23 12:07:23
local-1443006044432	Spark shell	2015/09/23 12:00:43	2015/09/23 12:01:58	1.3 min	robineast1	2015/09/23 12:01:58

图 9.14 Spark 的 history server UI 有两个已完成的 Spark 应用。

也可以写事件日志到其他目录，不用默认的 /tmp 目录，添加如下配置选项到 `spark-shell`：

```
--conf "spark.eventLog.dir=/logs/test-events"
```

这样启动 history server：

```
./sbin/start-history-server.sh /logs/test-events
```

也可以写事件日志到 HDFS（分布式数据存储层，这也是存储应用日志的最佳选择），路径前缀用 `hdfs:///`（如：`hdfs:///logs/spark-events`）。

提示：如果配置了 spark-shell 或 spark-submit，写日志到一个不存在的目录中，在 SparkContext 初始化时会报 NullPointerException 异常，导致应用失败。所以，要是遇到配置了事件日志目录而 Spark 应用没有启动的情况，检查一下日志目录是否已经创建。

9.2 Spark配置

Spark 这类并行处理系统的主要目的是尽可能充分利用 CPU 的处理能力，并保证每个 executor 都有足够的内存可用。要是只有一个用户独占集群资源，那很容易充分发挥 Spark 的能力。但要是多用户共享集群资源，就需要合理规划利用集群资源了。本节将介绍实现这些目标的配置选项。

Spark 有三种不同的集群部署模式：

- Standalone
- Mesos
- YARN

本书采用 Spark 自带的 Standalone 集群管理模式。要是选择 Mesos 或 YARN 管理集群，其大部分的配置与 Standalone 类似，但也需要与你的集群管理员一起确定这些相关的配置。

注意：Spark 文档中的名词“集群管理”通常表示，提供集群资源给 Spark 应用的服务（或一组服务）。Standalone 集群管理和 Spark Master 的含义相同，两者可互换使用。

下载了开箱即用的 Spark，运行不带参数的 bin/spark-shell，会启动一个非集群环境，即 driver 和一个 executor 运行在同一个 Java 虚拟机中。并且会默认分配 512MB 内存，申请这台机器上所有的 CPU 给 Spark 使用，这些内存和 CPU 资源被 driver 和 executor 共享。

虽然这种安装程序可以用于小数据集和初始的探索性工作，但要是用到大数据集，就需要在多节点集群上运行 Spark 作业。Spark 集群的规模可以从几台机器到数百台乃至数千台机器的一个大集群。集群中的每台机器通常会运行一个独立的 worker JVM 进程，它可以在当前这台机器上运行一个或多个执行器的 JVM。正如

所见，这就是运行请求任务的执行器。

为充分利用Spark集群，一般通过spark-submit或spark-shell启动driver程序，并提供如下的集群URL（<master-host>是运行Spark Master的机器名和端口号）：

```
bin/spark-shell --master spark://<master-host>:7077
```

Spark Master会从worker节点上提供集群资源（以executor进程的形式）。如果在上述命令行后面不带任何参数，会在每个worker节点上启动executor执行器，每个executor都会默认申请当前机器的全部CPU计算资源以及1GB的随机访问内存。一般情况下，想在每个worker节点上使用尽可能多的内存资源（毕竟这是Spark的主要优点，内存计算），可以加一些额外参数来指定每个worker节点使用的内存。我们假定每个worker节点都有32GB内存，设置--executor-memory参数，在每台worker节点上申请31GB内存（留给操作系统1GB内存）。

```
spark-shell --master spark://:7077 --executor-memory 31g
```

如图9.15所示，用参数--executor-memory配置4节点规模的集群。

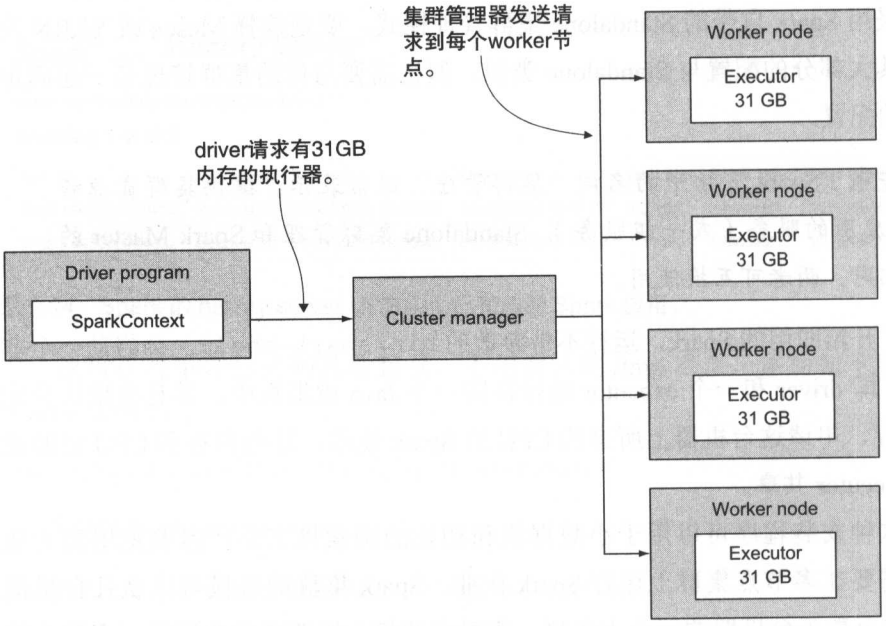


图9.15 集群中的每个执行器都申请了31GB内存。

其实，并不推荐使用太大的 JVM 堆，最好不大于 64GB，因为 JVM 垃圾回收耗时会变大。但是，多数情况下用于生产环境的 Spark 集群的机器配置都比较高，有 256GB、512GB 甚至更大的内存配置。为防止 executor 的 JVM 堆大于 64GB，可以在每个 worker 节点上申请多个小的执行器，合理配置 `--executor-memory` 和 `--executor-cores` 这两个参数。

假设 worker 节点有 256GB 内存和 24 核 CPU，理想情况下，在每个 worker 节点上运行 4 个 executor，每个 executor 限制 63GB 内存和 6 核 CPU：

```
spark-shell --master spark://:7077 --executor-memory 63g --executor-cores 6
```

从图 9.16 中可以看到这个新配置。注意，每个 executor 分配了 63GB 内存，不是 64GB；Spark 允许最多申请 255GB 内存（256GB 减去 1GB 给操作系统使用）。

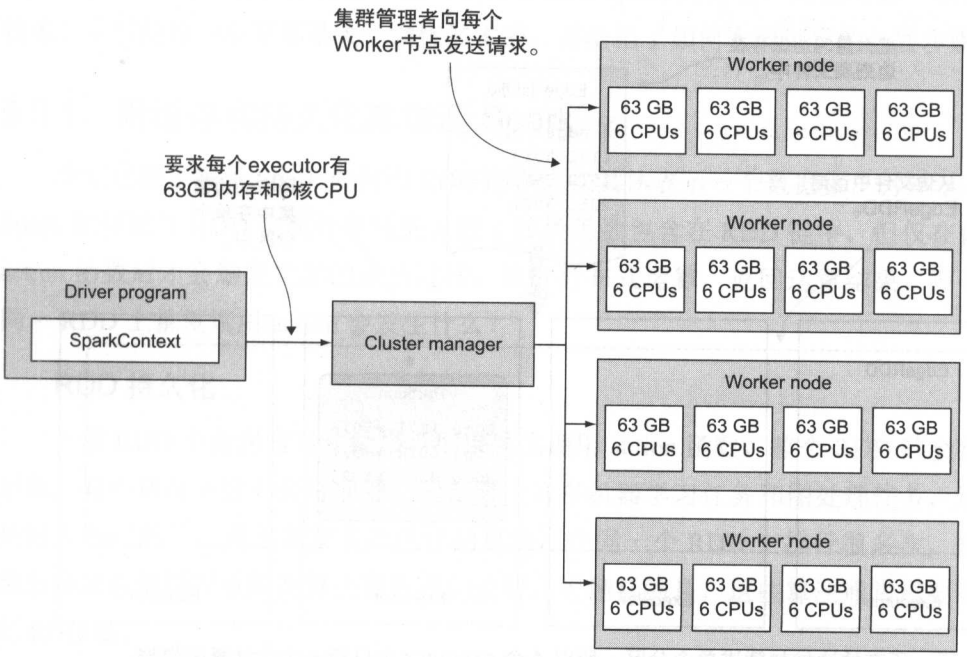


图 9.16 集群中每个 worker 节点上运行多个 63GB 的执行器。

Spark 小贴士：在配置 executor 的内存和 CPU 时，可以访问 Master 和 worker 的 Web 页面，Master 的端口是 8080，worker 的端口是 8081。这两类 Web 界面会详细地分别列出集群和每个 worker 可用的内存和 CPU 资源。

9.2.1 充分利用全部 CPU 资源

现在我们已经把集群配置为可以使用最多可用的 CPU 和内存资源，但仍需要进一步确保充分利用它们。下一节介绍内存的使用、查看缓存和持久性。本节着眼于最大限度地提高 CPU 的使用率。

在 `spark-shell` 或 `spark-submit` 上调整一些参数，可以确保集群中的应用有足够的内存和 CPU 可用，却不能保证这些可用的 CPU 和内存会被实际使用。

正如所见，Spark 通过独立处理每个分区来完成 stage 处理。事实上，一个 executor 对应一个分区，所以分区数要是小于 executor 的数目那就不能充分利用全部的资源了，如图 9.17 所示。

提示：可以调用 `RDD.partitions.size` 看 RDD 有多少分区。

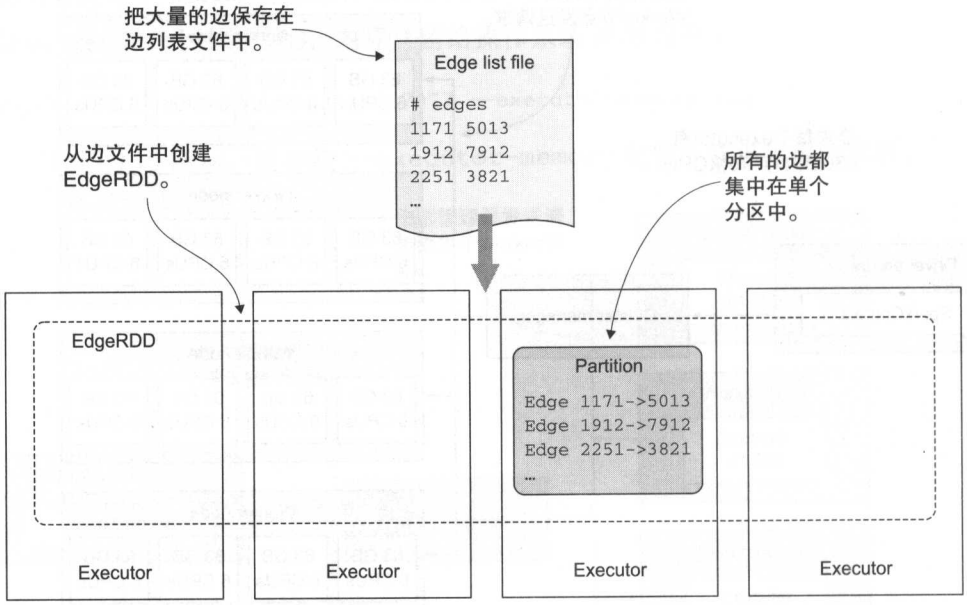


图 9.17 所有的边被加载进单个分区，所以 4 个 executor 中只有一个会计算图数据。

在这种情况下，大量的顶点和边会挤在一个 Spark 分区上。解决这个问题的方法是调用 RDD 的 `repartition` 方法，提供一个合适的新分区数给 `repartition` 函数作为参数。

```
val rdd = ...
rdd.repartition(20)
```

什么决定了分区数？我们看到这些 RDD 是建立在这些转换处理上的，所以 RDD 的分区数是基于其父 RDD 的分区数。

最终会追溯到一个没有父 RDD 的根 RDD。这类根 RDD 一般是从文件或数据库存储读数据。假设从 HDFS 读数据，RDD 分区数将会由 HDFS 文件块的数目决定。

一个通用的规则是，要确保至少有和 CPU 核数一样多的分区，分区两三倍或多倍于 CPU 更好，这是因为相比较于 Hadoop，Spark 的调度延迟比较低。

9.3 Spark性能调优

到目前为止，介绍的内容大都集中在如何执行应用程序代码和如何配置 Spark，以便有效地利用可用计算资源。但为了充分利用 Spark 的特性，应该考虑一些调优技术。本节的每一小节都是关于调优技术的，并给出了如何实现调优的必要步骤。

9.3.1 用缓存和持久化来加速 Spark

我们已经了解了 Spark 如何用 RDD 的概念来具体表示一个数据集，另外学习了 Spark 如何通过 RDD 实现计算链的原理：转换函数包含在 RDD 链中，但仅在调用 action 函数后才会触发实际的求值过程，执行分布式运算，返回运算结果。要是在同一 RDD 上重复调用 action 会发生什么？

RDD 持久化

一般 RDD 不会保留运算结果，如果再次调用 action 函数，整个 RDD 链会重新运算。有些情况下这不会有问题，但是对于许多机器学习任务和图处理任务，这就是很大的问题了。通常需要多次迭代的算法，在同一个 RDD 上执行很多次，反复地重新加载数据和重新计算会导致时间浪费。更糟糕的是，这些算法通常需要很长的 RDD 链。

看来我们需要另一种方式来充分利用集群可用内存来保存 RDD 的运算结果。这就是 Spark 缓存（缓存也是 Spark 支持的一种持久化类型）。

要在内存中缓存一个 RDD，可以调用 RDD 对象的 cache 函数。以下在 spark-shell 中执行的代码，会计算文件的总行数，输出文件内容：

```
val filename = "..."
val rdd1 = sc.textFile(filename).cache
```

```
rdd1.count
rdd1.collect
```

如果不调用 cache 函数，当 count 和 collect 这两个 action 函数被调用时，会导致执行从存储系统中读文件两次。调用了 cache 函数，第一个 action 函数（count 函数）会把它的运算结果保留在内存中，在执行第二个 action 函数（collection 函数）时，会直接在使用缓存的数据上继续运算，而不需要重新计算整个 RDD 链。

即使通过转换缓存的 RDD，生成新的 RDD，缓存的数据仍然可用。下面的代码会找出所有的注释行（以 # 开始的行数据）。

```
val rdd2 = rdd1.filter(_.startsWith("#"))
rdd2.collect
```

因为 rdd2 源于已缓存的 rdd1，rdd1 已经把它的运算结果缓存在内存中了，所以 rdd2 也就不需要重新从存储系统中读取数据。

注意：cache 方法作为一个标志表示 RDD 应当缓存，但并不是立即缓存。缓存发生在当前 RDD 在下一次要被计算的时候。

持久化等级

如上所述，缓存是其中一种持久化类型。表 9.3 列出了 Spark 支持的所有持久化等级。

表 9.3 Spark 支持的常用持久化等级

等级	描述
MEMORY_ONLY	将 RDD 作为 Java 对象存储在 JVM 中。如果内存不够 RDD 使用，一些分区就放弃缓存，这些未缓存分区会在每次需要的时候重计算。这也是默认的存储等级
MEMORY_AND_DISK	将 RDD 作为 Java 对象存储在 JVM 中。如果内存不够 RDD 使用，未在内存缓存的分区存储在磁盘上，需要的时候从磁盘读取
MEMORY_ONLY_SER	将 RDD 存储为序列化的 Java 对象。通常比未序列化的对象省空间，但它是 CPU 密集型读写方式
MEMORY_AND_DISK_SER	类似于 MEMORY_ONLY_SER 存储等级，RDD 作为序列化的 Java 对象存储，如果内存放不下整个 RDD，一部分分区被从内存中强制逐出，写入磁盘，这部分分区在需要时从磁盘读取
DISK_ONLY	RDD 分区保存在磁盘上，而不是内存中。如果 RDD 比可用内存大得多，这种存储等级就很有用，因为从本地磁盘读数据比从数据源读数据快

每个持久化等级都定义在单例对象 StorageLevel 中。例如，调用 rdd.persist

(StorageLevel.MEMORY_AND_DISK) 方法会把RDD设置成内存和磁盘缓存。cache方法内部也是调用rdd.persist(StorageLevel.MEMORY_ONLY)。9.3.3节会更详细地介绍MEMORY_ONLY_SER和MEMORY_AND_DISK_SER。

注意：其他的持久化等级，如MEMORY_ONLY2、MEMORY_AND_DISK2等，也是可用的。它们会复制RDD到集群的其他节点上，以便提供容错能力。这些内容超出了本书范围，感兴趣的读者可以看看Petar Zečević和Marko Bonaći（Manning, 2016）的书*Spark in Action*，这本书更深入地介绍了Spark容错方面的内容。

图的持久化

前面章节介绍过Spark从顶点RDD和边RDD构建出一个Graph对象，无论什么时候，通过Graph对象调用一些函数如mapVertices或aggregateMessages，这些操作都是基于下层的RDD实现的。

Graph对象提供了基于顶点RDD和边RDD方便的缓存和持久化方法。

在合适的时机反持久化

虽然看起来缓存是一个应该被到处使用的好东西，但是用得太多也会让人过度依赖它。

当缓存越来越多的RDD后，可用的内存就会减少。最终Spark会把分区数据从内存中逐出（使用最少最近使用算法，LRU）。同时，缓存过多的Java对象，JVM垃圾回收高耗是不可避免的。这就是为什么当缓存不再被使用时很有必要调用unpersist方法。对迭代算法而言，在循环中常用下面的方法调用模式：

- 调用Graph的cache或persist方法。
- 调用Graph的action函数，触发Graph下面的RDD被缓存……
- 执行算法主体的其余部分。
- 在循环体的最后部分，反持久化，即释放缓存。

你将在下一节介绍checkpoint时看到这样的一个例子。

提示：用Pregel API的好处是，它已经在内部做了缓存和释放缓存的操作。

何时不用缓存

不能盲目地在内存中缓存RDD。要考虑数据集会被访问多少次以及每次访问时重计算和缓存的代价对比，重计算也可能比增加内存的方式付出的代价小。

毫无疑问，如果仅仅读一次数据集，缓存RDD就毫无意义，这还会让作业运行得更慢，特别是用了有序列化的持久化等级。

9.3.2 checkpointing

图算法中一个常用的模式是用每个迭代过程中运算后的新数据更新图。这意味着，实际构成图的顶点RDD亦或边RDD的链会变得越来越长。

> **定义：**当RDD由逐级继承的祖先RDD链形成时，我们说从RDD到根RDD的路径是其谱系。

清单9.2所示的示例是一个简单的算法，可生成一个新顶点集并更新图。这个算法迭代的次数由变量iterations控制。

清单9.2 一个简单的迭代图更新算法

```
val iterations = 500
var g = Graph.fromEdges (sc.makeRDD(
          Seq(Edge(1L,3L,1),Edge(2L,4L,1),Edge(3L,4L,1))),1)
for (i <- 1 to iterations) {
   println("Iteration: " + i)
   val newGraph: Graph[Int, Int] =
       g.mapVertices((vid,vd)  => (vd * i)/17)
       g = g.outerJoinVertices[Int, Int](newGraph.vertices) {
                        (vid, vd, newData) => newData.getOrElse(0)
       }
}
g.vertices.collect.foreach(println)
```

上述代码每一次调用joinVertices都会增加一个新RDD到顶点RDD链中。

显然我们需要使用缓存来确保在每次迭代中避免重新计算RDD链，但这并不能改变一个事实，那就是有一个不断增长的子RDD到父RDD的对象引用列表。

这样的后果是，如果运行迭代次数过多，运行的代码中最终会爆出StackOverflowError栈溢出错误。通常迭代500次就会出现栈溢出。

而由 RDD 提供并且被 Graph 继承的一个特性：checkpointing，能解决长 RDD 谱系问题。清单 9.3 中的代码示范了如何使用 checkpointing，这样就可以持续输出顶点，更新结果图。

清单 9.3 简单的有 checkpointing 的迭代式图更新算法

```
sc.setCheckpointDir("/tmp/spark-checkpoint")
var updateCount = 0
val checkpointInterval = 50

def update(newData: Graph[Int, Int]): Unit = {
  newData.persist()
  updateCount += 1
  if (updateCount % checkpointInterval == 0) {
    newData.checkpoint()
  }
}

val iterations = 500
var g = Graph.fromEdges (sc.makeRDD(Seq(Edge(1L,3L,1),
                        Edge(2L,4L,1),Edge(3L,4L,1))),1)
update(g)
g.vertices.count
for (i <- 1 to iterations) {
  println("Iteration: " + i)
  val newGraph: Graph[Int, Int] =
                    g.mapVertices ((vid,vd) => (vd * i)/17)
  g = g.outerJoinVertices[Int, Int](newGraph.vertices) {
  (vid, vd, newData) => newData.getOrElse(0) }
  update(g)
  g.vertices.count
}
g.vertices.collect.foreach(println)
```

设置checkpoint目录，如果不设置，那么随后的checkpoint不会有任何效果。

记录更新了多少次图。

每50次图更新做一次checkpoint。

这个更新函数在每次更新图的时候调用，其中缓存和checkpoint是必需的。

在每次更新图后调用update方法，然后执行具体的算法实现，否则就没有更新和checkpoint的内容了。

一个标记为 checkpointing 的 RDD 会把 RDD 保存到一个 checkpoint 目录，然后指向父 RDD 的连接被切断，即切断了 lineage 谱系。一个标记为 checkpointing 的 Graph 会导致下面的顶点 RDD 和边 RDD 做 checkpoint。

调用 SparkContext.setCheckpointDir 来设置 checkpoint 目录，指定一个共享存储系统的文件路径，如 HDFS。

如前面的代码清单所示，必须在调用 RDD 任何方法之前调用 checkpoint，这是因为 checkpointing 是一个相当耗时的过程（毕竟需要把图写入磁盘文件），通常需要不断地 checkpoint 避免栈溢出错误，一般可以每 100 次迭代做一次 checkpoint。

> **注意**：一个加速 checkpointing 的选择是 checkpoint 到 Tachyon（已更名为 Alluxio），而不是 checkpoint 到标准的文件系统。Alluxio，来自 AMPLab，是一个“以内存为中心的有容错能力的分布式文件系统，它能让 Spark 这类集群框架加速访问共享在内存中的文件”。

9.3.3 通过序列化降低内存压力

内存压力（内存不够用）往往是 Spark 应用性能差和容易出故障的主要原因之一。这些问题通常表现为频繁的、耗时的 JVM 垃圾回收和“内存不足”的错误。checkpointing 在这里也不能缓解内存压力。遇到这种问题，首先要考虑序列化 Graph 对象。

> **定义**：数据序列化，Data serialization，是把 JVM 里表示的对象实例转换（序列化）成字节流；把字节流通过网络传输到另一个 JVM 进程中；在另一个 JVM 进程中，字节流可以被“反序列化”为一个对象实例。Spark 用序列化的方式，可以在网络间传输对象，也可以把序列化后的字节流缓存在内存中。

要用序列化，可以选用 persist 中下面的 StorageLevels：

- StorageLevel.MEMORY_ONLY_SER
- StorageLevel.MEMORY_AND_DISK_SER

序列化节省了空间，同时序列化和反序列化也会增加 CPU 的开销。

使用 Kryo 序列化

Spark 默认使用 JavaSerializer 来序列化对象，这是一个低效的 Java 序列化框架，一个更好的选择是选用 Kryo。Kryo 是一个开源的 Java 序列化框架，提供了快速高效的序列化能力。

Spark 中使用 Kryo 序列化，只需要设置 spark.serializer 参数为 org.apache.spark.serializer.KryoSerializer，如这样设置命令行参数：

```
spark-shell --conf
➥ "spark.serializer=org.apache.spark.serializer.KryoSerializer"
```

要是每次都这样设置参数，会很烦琐。可以在 $Spark_HOME/conf/spark-defaults.conf 这个配置文件中，用标准的属性文件语法（用 Tab 分隔作为一行），

把 spark.serializer 等参数及其对应的值写入这个配置文件，如下所示：

```
spark.serializer org.apache.spark.serializer.KryoSerializer
```

为保证性能最佳，Kryo 要求注册要序列化的类，如果不注册，类名也会被序列化在对象字节码里，这样对性能有较大影响。幸运的是，Spark 对其框架里用到的类做了自动注册；但是，如果应用程序代码里有自定义的类，恰好这些自定义类也要用 Kryo 序列化，那就需要调用 SparkConf.registerKryoClasses 函数来手动注册。下面的清单 9.4 展示了如何注册 Person 这个自定义类。

清单 9.4 使用 Kryo

```
import org.apache.spark.storage.StorageLevel
case class Person(name: String, age: Int)
val conf = new SparkConf()
conf.set("spark.serializer",
    "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[Person]))
val sc = new SparkContext (conf)
val rdd = sc.makeRDD(1 to 1000000).
          map(el => Person("John Smith", 42))
rdd.persist(StorageLevel.MEMORY_ONLY_SER)
rdd.count
```

定义Person类。

用SparkConf对象设置Spark配置参数。

配置必需的Kryo序列化参数。

向Kryo注册自定义类列表。

检查 RDD 大小

在应用程序调优时，常常需要知道 RDD 的大小。这就很棘手，因为文件或数据库中对象的大小和 JVM 中对象占用多少内存没有太大关系。

一个小技巧是，先将 RDD 缓存到内存中，然后到 Spark UI 中的 Storage 选项卡，这里记录着 RDD 的大小。要衡量配置了序列化的效果，用这个方法也可以。

9.4 图分区

第 1 章提到过图分区策略，这是 GraphX 的一个优点，即如何采用顶点切分法（把边按组划分）代替简单的边切分法进行分区（把顶点按组划分）。而第一次构建一个图时，要么用 Graph.apply()（与 Graph() 等价），要么用 GraphLoader，而图是“未分区”的。EdgeRDD 和 VertexRDD 有各自标准的分区方法，但图整体上并不以任何逻辑形式分区。

这会导致性能很差，此外，一些函数如 groupEdges() 和 triangleCount()

要求图分区才能正确运算。

传入参数 PartitionStrategy，调用 partitionBy() 函数可以对图进行分区。API 中的 PartitionStrategy 接口有 4 个实现类，即提供了如下 4 个分区策略，如图 9.18 所示。

- RandomVertexCut：通常这是最好的划分策略，代替边切分法的两个分区策略之一，最佳的负载均衡，但网络通信成本较大。
- CanonicalRandomVertexCut：与 RandomVertexCut 一样都是点切分法，不同的是，它能保证任何两点之间的重复边被划分到同一分区。但是，如果图中不存在重复边，一个需要访问处于分区边界上的边属性的算法，其性能就会有问题了。
- EdgePartition1D：保证一个顶点的边被划分在同一个分区上。
- EdgePartition2D：支持边邻接矩阵，并将其划分成片。它的缺点是，分区数是平方数（4、9、16，等等）。如果分区的个数不是一个平方数，就取与它最接近的较小的平方数作为分区数，从而导致负载不均衡问题。

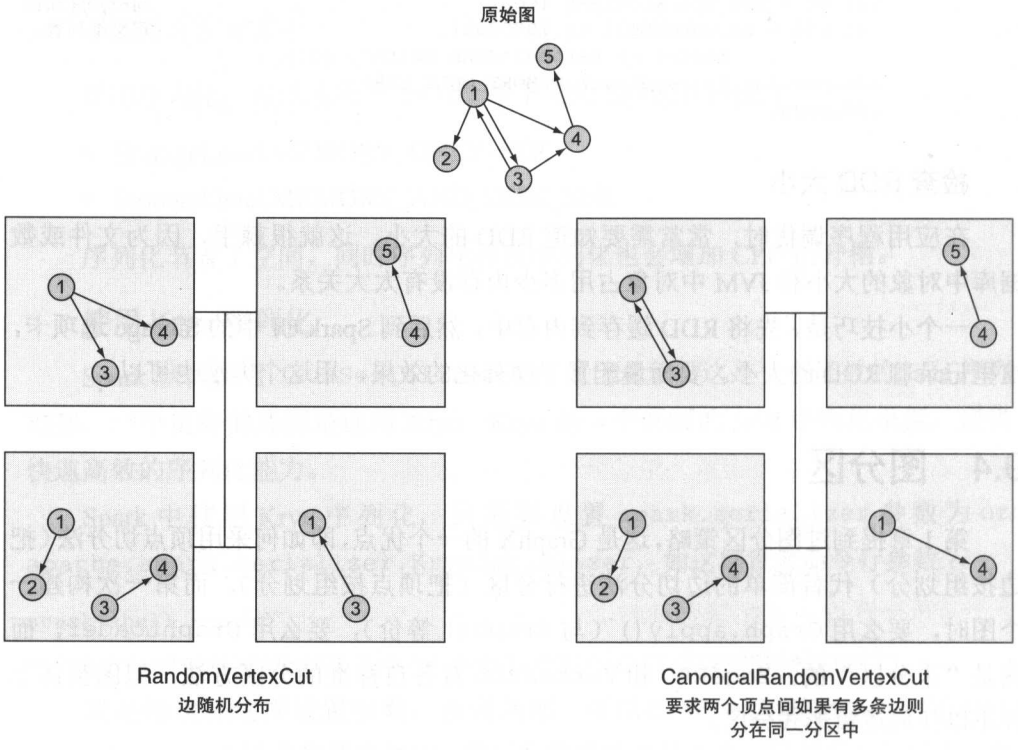


图 9.18 4 种不同的图分区策略选项，示例图的 4 种分区形式。

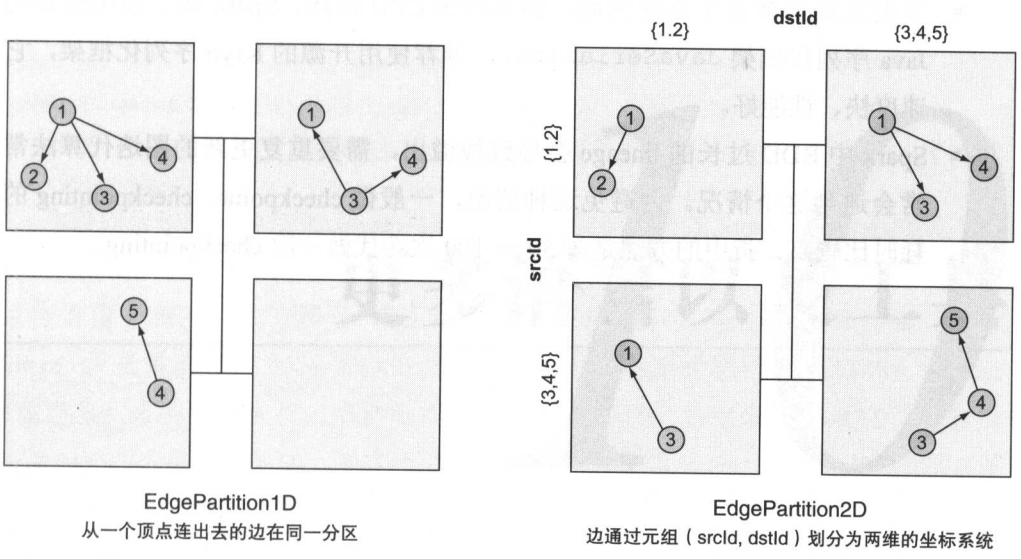


图 9.18 4 种不同的图分区策略选项，示例图的四种分区形式（续）。

由于顶点是独立划分的，所以并不能保证边的两个顶点会被划分在同一分区（事实上，这是非常不可能的）。通常，Spark 不得不序列化边和顶点生成一个 `EdgeTriplets`。`PartitionStrategy` 分区策略只会影响边，也允许在负载均衡（两种随机策略）和访问邻边的效率（两种边划分策略）之间做折中。

9.5 小结

- Spark 应用实际是执行一系列作业，这些作业被切分为不同的 stage，每个 stage 创建可执行的 task 任务来处理 RDD 一个分区上的数据。
- 熟悉 Spark UI，了解 Spark 应用是如何执行的。作业列表页和详情页提供了作业、stage、task 任务的开始时间和持续时间的统计信息。事件时间线提供了可视化工具来发现应用的性能问题。
- 可调优的地方包括：缓存、checkpointing、序列化和分区。
- 同一份数据被频繁查询时，缓存的作用就凸显了。通过在内存缓存数据，Spark 避免每次从数据源重新获取数据，重新计算。不同的缓存等级可以控制缓存在不同的地方（内存或磁盘）以及缓存的数据是否需要序列化。

- 序列化数据节省了内存空间，但会增加 CPU 消耗。Spark 默认使用低效的 Java 序列化框架 `JavaSerializer`，推荐使用开源的 Kryo 序列化框架，它速度快、性能好。
- Spark 中 RDD 过长的 lineage 会导致栈溢出，需要重复更新的图迭代算法常常会遇到这个情况。为避免这种情况，一般做 checkpoint。checkpointing 的耗时比较高，折中的方法是每 50 ～ 100 次迭代做一次 checkpointing。

10 更多语言以及工具

本章要点

- 在 GraphX 中使用 Java 7 和 Java 8 来代替 Scala
- GraphX 中的 Python 和 R 接口现状
- 作为 Gephi 的替代方案，使用 Apache Zeppelin 和 d3.js 进行图的可视化
- 使用 Spark Job Server 获得类似数据库的能力
- 使用 GraphFrames

到目前为止，我们都是使用 Scala 语言进行编程，使用 Gephi 进行可视化的。在本章，我们会介绍 Spark 支持的其他语言，但是，GraphX 对这些语言的支持也是有限的。同时，我们还会介绍如何使用 d3.js 和 Apache Zeppelin 进行图的可视化，它们是类似 Mathematica、IPython Notebook 和 Jupyter 的 notebook 软件。notebook 软件的优点在于结合了类似 REPL 的交互形式以及可视化。

在本书中我们已多次重申 GraphX 不是一个图数据库。然而，使用额外的 Spark Job Server 工具，在 Spark 上添加相应的 REST 接口，我们几乎可以将 GraphX 变成一个相当轻量级的数据库。

最后，我们会介绍库 GraphFrames 的应用，这也是由 GraphX 的部分开发者提出的，它使得图的查询更为简单以及高效。

10.1 在GraphX中使用除Scala外的其他语言

尽管 Scala 是 Spark 的原生语言，但仍有使用 Spark 支持的其他语言的需求：个人偏好、团队或企业偏好、安全性、和已有库的兼容性等。在 Spark 1.6 的发布版本中，我们可以在 GraphX 中使用 Java 7 或 Java 8，但仍未支持 Python 和 R 语言，Java 的 API 接口使用起来也并不方便。Jira 上 SPARK-3665 的主题就是为 GraphX 创建真正的 Java 友好接口，但还没被 Spark 1.6 所接纳。你可能会因为众多理由而想在 GraphX 中使用 Java，但请注意使用起来并不方便。如果仅仅是因为相比 Scala 更熟悉 Java 语言而做出这个决定，那么我建议你多考虑考虑。相比使用 Scala，使用 Java 时你可能需要写十倍的代码量，包含大量的模糊构建。此外，使用 Java 时不能在 REPL 中运行。

以下两个使用 Java 的原因会相对合理：

- 公司要求。
- 与类似 Fortify 的字节编码工具的兼容性，它们不能很好地处理由 Scala 生成的大量 .class 文件。

10.1.1 节主要讲述在 GraphX 中使用 Java 7 时可能会遇到的各种难以处理的问题。Java 8 的 lambda 表达式也不会给 GraphX 带来太多的便利性，在 10.1.2 节会介绍 Java 8 的 lambda 表达式可以发挥有限作用的地方。10.1.3 节会介绍如何在 GraphX 中使用 Python 和 R，但它们依旧不属于 Apache Spark 1.6 的官方版本的内容。

10.1.1 在 GraphX 中使用 Java 7

在本小节，我们以 4.2.3 小节的边个数统计作为例子，将它转为使用 Java 实现。由于 REPL 环境不支持 Java 的运行，我们首先需要创建用于 Maven 工程的 pom.xml 文件。

清单 10.1 中的 pom.xml 文件是一个通用的版本。`exec-maven-plugin` 中有一个巧妙的设置：将 `cleanupDaemonThreads` 设置为 `true`。这允许程序使用 `mvn exec:java` 命令来执行，当程序关闭后 Maven 也会将所有 Spark 线程同时结束。

清单 10.1 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning</groupId>
  <artifactId>graphx-propagate-edge-count</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.5.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-graphx_2.10</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <executions>
          <execution>
            <goals>
              <goal>java</goal>
            </goals>
          </execution>
        </executions>
```

```
        <configuration>
          <mainClass>EdgeCount</mainClass>
          <cleanupDaemonThreads>false</cleanupDaemonThreads>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

清单10.2中的Java代码太长了。一部分代码包含了大量的引用，用于解决Java/Scala的互通性问题；一部分代码采用了更为烦琐的句法来构建示例的myVertices和myEdges变量。但其中也包含一部分高效的代码来处理lambda表达式。函数propagateEdgeCount()的最后部分，采用了与Spark Core相关的lambda表达式来调用map()和reduce()。这部分的代码还比较可取。这是因为Apache Spark的开发者们，为了可以简单方便地使用lambda表达式来处理Spark Core而费尽心思。但他们没有为GraphX做出类似的优化。

用于处理Spark Core的lambda表达式是o.a.s.api.java.Function、o.a.s.api.java.Function2等的实例（o.a.s.是org.apache.spark的缩写），这里，Function代表输入一个参数的匿名函数，Function2代表输入两个参数，依此类推。

相反，用于Spark GraphX的lambda表达式则是scala.Function1、scala.Function2等的实例。如果你查看了GraphX的Javadocs文档（而不是常规的ScalaDocs），会在Graph.mapVertices()等函数上看到这些鲜明的特性。但若想在Java上直接举例说明Function1、Function2等的用法，几乎是不可能的。因为这时你需要重载数十个相关的函数（大部分的函数名字里包含了$符号）。因此，Scala库为Java程序员提供了scala.runtime.AbstractFunction1、scala.runtime.AbstractFunction2等方便使用的类。在使用这些类时，你只需重载感兴趣的apply()函数。但是，仅做到上述操作仍是不够的，因为Spark不仅采用了函数式编程，更是分布式的，所以在Spark中使用的lambda表达式也需要是可串行化的，但是AbstractFunction1、AbstractFunction2等并不满足这个要求。在下述代码的开始部分，我们定义了SerializableFunction1和SerializableFunction2来替代AbstractFunction1和AbstractFunction2等类。

清单 10.2 EdgeCount.java

```
import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

import scala.Tuple2;
import scala.reflect.ClassTag;
import scala.reflect.ClassTag$;
import scala.runtime.AbstractFunction1;
import scala.runtime.AbstractFunction2;
import scala.runtime.BoxedUnit;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.graphx.*;
import org.apache.spark.rdd.RDD;
import org.apache.spark.storage.StorageLevel;

public class EdgeCount {
  // 提供给 aggregateMessages() 函数的 sendMsg 和 mergeMsg 这两个函数
  // 需要既是 Scala 的（GraphX API 要求）又是可序列化的（Spark 要求）
  static abstract class SerializableFunction1<T1,R>
    extends AbstractFunction1<T1,R> implements Serializable {}

  static abstract class SerializableFunction2<T1,T2,R>
    extends AbstractFunction2<T1,T2,R> implements Serializable {}

  public static void main(String[] args) {
    JavaSparkContext sc = new JavaSparkContext(
      new SparkConf().setMaster("local").setAppName("EdgeCount"));

    JavaRDD<Tuple2<Object, String>> myVertices =
      sc.parallelize(Arrays.asList(new Tuple2<Object,String>(1L, "Ann"),
```

```
        new Tuple2<Object,String>(2L, "Bill"),
        new Tuple2<Object,String>(3L, "Charles"),
        new Tuple2<Object,String>(4L, "Diane"),
        new Tuple2<Object,String>(5L, "Went to gym this morning")));

    JavaRDD<Edge<String>> myEdges = sc.parallelize(Arrays.asList(
      new Edge<String>(1L, 2L, "is-friends-with"),
      new Edge<String>(2L, 3L, "is-friends-with"),
      new Edge<String>(3L, 4L, "is-friends-with"),
      new Edge<String>(4L, 5L, "Likes-status"),
      new Edge<String>(3L, 5L, "Wrote-status")));

    Graph<String,String> myGraph = Graph.apply(myVertices.rdd(),
      myEdges.rdd(), "", StorageLevel.MEMORY_ONLY(),
      StorageLevel.MEMORY_ONLY(), tagString, tagString);

    Graph<Integer,String> initialGraph = myGraph.mapVertices(
      new SerializableFunction2<Object,String,Integer>() {
        public Integer apply(Object o, String s) { return 0; }
      },
      tagInteger, null);

    List<Tuple2<Object,Integer>> ls = toJavaPairRDD(
      propagateEdgeCount(initialGraph).vertices(), tagInteger).collect();

    for (Tuple2<Object,Integer> t : ls)
      System.out.print(t + " ** ");

    System.out.println();

    sc.stop();
  }

  // 必须在各种函数调用中显式提供隐式 Scala 参数
  private static final ClassTag<Integer> tagInteger =
    ClassTag$.MODULE$.apply(Integer.class);
```

```
private static final ClassTag<String> tagString =
  ClassTag$.MODULE$.apply(String.class);
private static final ClassTag<Object> tagObject =
  ClassTag$.MODULE$.apply(Object.class);

// sendMsg
private static final SerializableFunction1<
    EdgeContext<Integer, String, Integer>, BoxedUnit> sendMsg =
  new SerializableFunction1<
      EdgeContext<Integer, String, Integer>, BoxedUnit>() {
    public BoxedUnit apply(EdgeContext<Integer, String, Integer> ec) {
      ec.sendToDst(ec.srcAttr()+1);
      return BoxedUnit.UNIT;
    }
  };

// mergeMsg
private static final SerializableFunction2<Integer, Integer, Integer>
  mergeMsg = new SerializableFunction2<Integer, Integer, Integer>() {
    public Integer apply(Integer a, Integer b) {
      return Math.max(a,b);
    }
  };

private static <T> JavaPairRDD<Object,T>
    toJavaPairRDD(VertexRDD<T> v, ClassTag<T> tagT) {
  return new JavaPairRDD<Object,T>((RDD<Tuple2<Object,T>>)v,
                                   tagObject, tagT);
}

private static Graph<Integer,String> propagateEdgeCount(
    Graph<Integer,String> g) {
  VertexRDD<Integer> verts = g.aggregateMessages(
    sendMsg, mergeMsg, TripletFields.All, tagInteger);
  Graph<Integer,String> g2 = Graph.apply(verts, g.edges(), 0,
    StorageLevel.MEMORY_ONLY(), StorageLevel.MEMORY_ONLY(),
```

```
      tagInteger, tagString);
    int check = toJavaPairRDD(g2.vertices(), tagInteger)
      .join(toJavaPairRDD(g.vertices(), tagInteger))
      .map(new Function<Tuple2<Object,Tuple2<Integer,Integer>>,
                        Integer>() {
        public Integer call(Tuple2<Object,Tuple2<Integer,Integer>> t) {
          return t._2._1 - t._2._2;
        }
      })
      .reduce(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer b) {return a+b;}
      });
    if (check > 0)
      return propagateEdgeCount(g2);
    else
      return g;
  }
}
```

以下是一些需要注意的 API 变更：

- 使用 `JavaSparkContext` 代替 `SparkContext`。
- 使用 `Object` 代替 `VertexId`。当需要比较值的大小时，你需要在相应的地方将 `Object` 强制转化为 `Long` 类型。
- 使用 `parallelize()` 代替 `makeRDD()`。
- `parallelize()` 仅对 `Lists` 适用，不可用于 `Arrays`。

由于 lambda 表达式总要求返回一个值，但 Java 里没有 `Unit` 类型，Scala 提供了单例的 `scala.runtime.BoxedUnit.UNIT` 作为替代。

Scala 的很多良好特性都不适用于 Java 语言，最先想到的就是默认参数的特性。在使用 GraphX 的 Java API 时，我们需要完整提供所有的参数。例如，调用 `Graph.apply()` 创建图时需要提供 7 个参数，而使用 Scala 时仅需提供 2 个参数。为了完整提供所有的参数，经常需要参考 ScalaDocs 来查看默认参数的内容。

一部分默认参数是 Scala 会自动提供的隐式 `ClassTags`。但使用 Java 时，你需要手工计算和提供这些 ClassTags。

另一个值得怀念的特性是 RDD 和 PairRDD 的自动转换（也即 Java 中的 JavaRDD 和 JavaPairRDD）。所以我们编写了一个帮助函数 toJavaPairRDD()，以便在需要的时候进行强制转换。

10.1.2 在 GraphX 中使用 Java 8

很多人认为 Java 8 为 Java 语言引进了类似 Scala 的函数式编程方式，但 Java 8 的 lambda 表达式只在处理 Spark Core 时有效。清单 10.2 里使用到的 Spark Core 的 lambda 表达式是 map() 和 reduce()，用于计算 check 变量。下面的清单 10.3 展示了如何使用 Java 8 来进行 check 变量的处理。

清单 10.3 将 EdgeCount.java 的部分实现转换成 Java 8 lambda 表达式

```
int check = toJavaPairRDD(g2.vertices(), tagInteger)
   .join(toJavaPairRDD(g.vertices(), tagInteger))
   .map(t -> t._2._1 - t._2._2)
   .reduce((a,b) -> a+b);
```

10.1.3 未来 GraphX 是否会支持 Python 或者 R

Jira SPARK-3789 主张要在 GraphX 中支持 Python，但在 Spark 1.6 版本中并未接受这一提议。至于对 R 语言的支持，在 2015 年 8 月 6 日就有 AMPLab 开发者在 Apache Spark 用户邮件中提议过，尽管将一些高层次算法（如 PageRank）的接口提供给 R 程序员会很有意义，但没有必要提供 GraphX 的所有 API 接口。10.4 节会介绍的 GraphFrames，就支持了 Python 语言。

10.2 其他可视化工具：Apache Zeppelin 和 d3.js

除了使用 Spark REPL 和 Gephi 两个独立的工具来进行图的可视化外，还可以组合使用 Apache Zeppelin 和 d3.js，这样我们可以兼得类似 REPL 交互式编程能力和强大的可视化能力。唯一的限制是，进行可视化处理时需要 d3.js 和 JavaScript 的相关知识，这超过了本书的范畴，但我们会提供一些示例代码。这里提供的是可简单快速启动的可视化方案，如果需要更复杂的操作你可以使用 Gephi 处理。

交互式编程的强大理念最初由 Mathematica 提出，后来被 IPython Notebook 仿效，

现在更为出名的是 Jupyter。Zeppelin 也是基于这一理念进行处理的，并属于 Spark 内置模块，使用起来更为简单方便。图 10.1 展示了如何在交互式编程环境中使用 Spark 命令进行可视化操作。

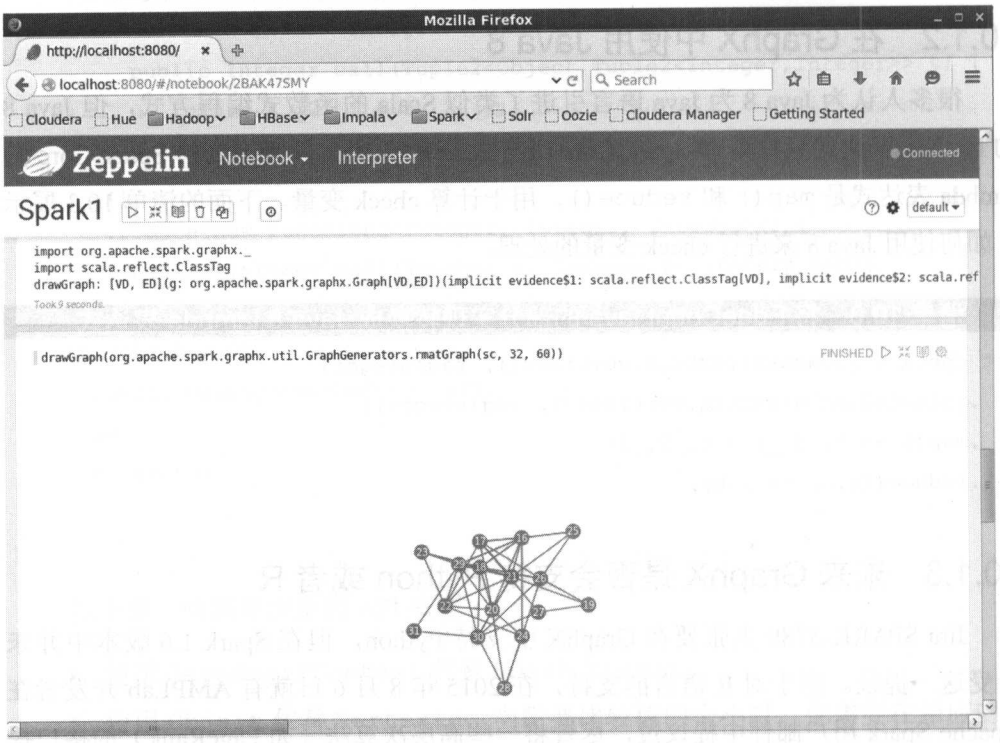


图 10.1 Zeppelin 与 Spark REPL 很相似，但它还可以在内部进行可视化展示。

首先，从 https://zeppelin.apache.org 下载 Zeppelin。然后在 Linux 终端使用以下命令（根据实际情况修改版本号）：

```
tar -xzvf zeppelin-0.5.6-incubating-bin-all.tgz
./zeppelin-0.5.6-incubating-bin-all/bin/zeppelin-daemon.sh start
xdg-open http://localhost:8080
```

最后一步是在端口 8080 开启一个网页浏览器。在浏览器页面上，单击“Create new note”（创建新的节点）。

和大部分交互式编程软件类似，Zeppelin 支持内联的 JavaScript，这意味着我们可以使用 d3.js 的可视化库进行处理。清单 10.4 展示了一些基本的可视化方法，对应的结果如图 10.1 所示。为了进行进一步的定制化（例如添加边标签、定位顶点标签，

以及在有向图中添加箭头等），你需要编写 JavaScript 代码来实现这些功能。可参考 Elijah Meeks 的书籍 *D3.js in Action*（manning，2015）。

清单 10.4 在 d3.js JavaScript 和 Scala 中实现 drawGraph

```
import org.apache.spark.graphx._
import scala.reflect.ClassTag
def drawGraph[VD:ClassTag,ED:ClassTag](g:Graph[VD,ED]) = {
val u = java.util.UUID.randomUUID
val v = g.vertices.collect.map(_._1)
println("""%html
<div id='a""" + u + """' style='width:960px; height:500px'></div>
<style>
.node circle { fill: gray; }
.node text { font: 10px sans-serif;
             text-anchor: middle;
             fill: white; }
line.link { stroke: gray;
            stroke-width: 1.5px; }
</style>
<script src="//d3js.org/d3.v3.min.js"></script>
<script>
var width = 960, height = 500;

var svg = d3.select("#a""" + u + """").append("svg")
    .attr("width", width).attr("height", height);

var nodes = [""" + v.map("{id:" + _ + "}").mkString(",") + """];
var links = [""" + g.edges.collect.map(
  e => "{source:nodes[" + v.indexWhere(_ == e.srcId) + "],target:nodes[" +
       v.indexWhere(_ == e.dstId) + "]}").mkString(",") + """];

var link = svg.selectAll(".link").data(links);
link.enter().insert("line", ".node").attr("class", "link");

var node = svg.selectAll(".node").data(nodes);
var nodeEnter = node.enter().append("g").attr("class", "node")
```

```
nodeEnter.append("circle").attr("r", 8);

nodeEnter.append("text").attr("dy", "0.35em")
        .text(function(d) { return d.id; });

d3.layout.force().linkDistance(50).charge(-200).chargeDistance(300)
   .friction(0.95).linkStrength(0.5).size([width, height])
   .on("tick", function() {
      link.attr("x1", function(d) { return d.source.x; })
          .attr("y1", function(d) { return d.source.y; })
          .attr("x2", function(d) { return d.target.x; })
          .attr("y2", function(d) { return d.target.y; });
      node.attr("transform", function(d) {
        return "translate(" + d.x + "," + d.y + ")";
      });
   }).nodes(nodes).links(links).start();
</script>
""")
}
```

如果将上述清单中的代码放在一个 Zeppelin 单元中，那么可以使用以下方法在另一个单元中进行测试：

```
drawGraph(org.apache.spark.graphx.util.GraphGenerators.rmatGraph(sc,32,60))
```

清单 10.4 中的代码混合使用了 Scala 和 JavaScript 语言，而为了将顶点和边的数据注入到 JavaScript 中，Scala 代码部分生成了相应的 JavaScript 代码。为了进行布局的调整，你可以仔细研究 `d3.layout.force()` 的参数设置，具体可参考 https://github.com/mbostock/d3/wiki/Force-Layout 上的文档。这里并不需要大量的 JavaScript 知识来调整颜色、画布大小等，但标签定位、添加箭头等则需要了解 d3.js 的专业知识。

10.3 类似一个数据库：Spark Job Server

在本书的介绍中，对 GraphX 的定位更多的是图处理系统，而不是一个数据库。它确实会分批处理图数据（以作业的方式），并最后输出一个结果。

但是，当越来越多的开发者和企业用户使用 Spark 时，将 GraphX 当成一个数据库来使用也是不错的选择。当然这个“数据库”不支持事务或锁等操作，但依然能充分满足用户的需求。

早在 2014 年，流式视频技术公司 Ooyala 开启了一个名为 Spark Job Server 的 GitHub 项目，用于处理共享的 RDD。Spark Job Server 值得用专门的章节来进行介绍，本章中我们将会介绍将它和 GraphX 结合使用的一个例子。

考虑到创建 Spark Job Server 的动机时，可以从新接触 Spark 的用户出发。他们经常会遇到一个问题：现在我有大量数据存储在 RDD 中，那怎么在多个应用中来共享这些数据呢？事实上一般情况下这是不可行的，因为 RDD 和特定的 `SparkContext` 绑定了，而 `SparkContext` 又和特定的 JVM 应用绑定了。你不可以共享 RDD 数据，除非使用 Spark Job Server 来处理（目前而言，至少在长期存在的 Jira SPARK-2389 得到解决之前，这是唯一的方法）。图 10.2 展示了如何在 Spark Job Server 中获得一个 `SparkContext`，并允许通过 REST 接口来调用这个 `SparkContext`。

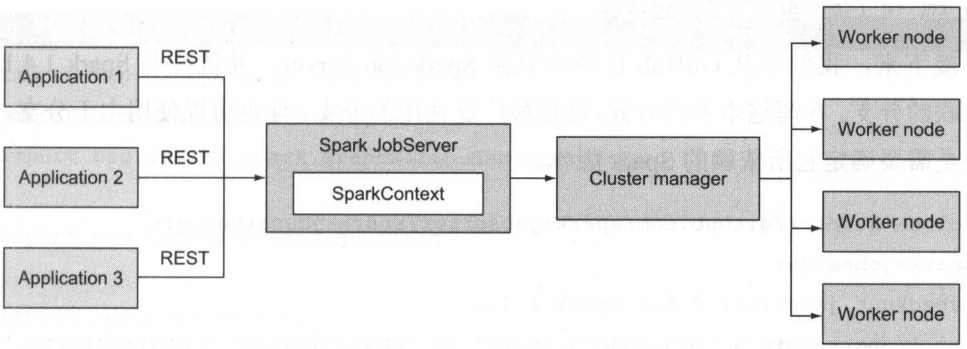


图 10.2 Spark Job Server 会保持 SparkContext 实例，因此也会保持与之相关联的 RDD 引用，允许多个应用共享同一 SparkContext 和 RDD 数据。图数据也可以加载到 Spark Job Server 中，多个应用可以对这个图进行“查询”。甚至还可以提供更新图的方式。

当一个 GraphX 的图被加载到 Spark Job Server 后，多个应用可以对这个图进行“查询”。如果只有一个应用，这种做法仍然是有效的，Spark 的 cluster 节点可以运行在 cluster 模式而不是 client 模式。cluster 模式与 client 模式的对比超出了本书的讨论范畴。但需要了解的是，在 cluster 模式下，应用并不会保持 `SparkContext` 实例。所以对于 cluster 模式下的 cluster 节点而言，Spark Job Server 会允许一个应用不断重用 RDD（或者是 Graph），而不需要提交新的 Spark 任务来从分布式存储中重新加载数据。

10.3.1 示例：查询 Slashdot 好友的分离程度

接下来，在使用 Spark Job Server 提供一个查询静态图数据的例子中，我们将会介绍如何安装和启动 Spark Job Server，如何创建一个用于被 Spark Job Server 加载的 .jar 文件，以及如何使用 REST 接口来进行调用。我们使用 5.2.2 节中提到的 SNAP 数据集——Slashdot 的好友以及敌人数据集，同时也提供了查询的能力，即提供两个 Slashdot 用的 ID，返回它们的分离程度（类似针对 Kevin Bacon 的六度理论，即凯文指数）。

安装和启动 Spark Job Server

首先，确保没有运行中的 Spark Job Server。在 Cloudera QuickStart VM 的老版本中，你可能会找到一个运行中的已废弃的 Spark Job Server。可以使用下述命令来终止它：

```
sudo pkill -f spark-jobserver
sudo rm -r /tmp/spark-jobserver
```

接下来，我们会从 GitHub 目录中克隆 Spark Job Server，并选择与 Spark 1.4.1 相关联的分支。在写这本书的时候，这是最广泛使用的分支。你也可以使用主干分支，前提是需要确定它所依赖的 Spark 版本。

```
git clone https://github.com/spark-jobserver/spark-jobserver.git
cd spark-jobserver
git checkout jobserver-0.6.0-spark-1.4.1
sed -i '/spark-core/a "org.apache.spark" %% "spark-graphx" % sparkVersion,'
➥ project/Dependencies.scala
sbt
restart
```

下载 Slashdot 数据

然后，下载 Slashdot 数据，将它放在主目录下。

```
cd ~
wget http://snap.stanford.edu/data/soc-Slashdot0811.txt.gz
gzip -d ~/soc-Slashdot0811.txt.gz
```

构建定制化的任务 Jar 包

我们使用 SBT 来构建定制化的任务 jar 包，如清单 10.5 所示。和本书其他基于 SBT 构建的项目一样，清单 10.6 中的 .scala 文件要存储在 src/main/scala 文件夹下。请确保 Spark 版本号和 Spark Job Server 版本号，应与早先所选择的 Spark Job Server 分支的版本号一致。

清单 10.5 sjsslashdot.sbt

```
scalaVersion := "2.10.4"
resolvers += "Job Server Bintray" at
➥ "https://dl.bintray.com/spark-jobserver/maven"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.1"
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "1.4.1"
libraryDependencies += "spark.jobserver" %% "job-server-api" % "0.6.0" %
➥ "provided"
```

清单 10.6 sjsslashdot.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.graphx._
import org.apache.spark.graphx.lib.ShortestPaths

import com.typesafe.config.Config

import spark.jobserver._

object Degrees extends SparkJob {
  val filename = System.getProperty("user.home") + "/soc-Slashdot0811.txt"
  var g:Option[Graph[Int,Int]] = None

  override def runJob(sc:SparkContext, config:Config) = {
    if (!g.isDefined)
      g = Some(GraphLoader.edgeListFile(sc, filename).cache)

    val src = config.getString("src").toInt

    if (g.get.vertices.filter(_._1 == src).isEmpty)
```

```
      -1
    else {
      val r = ShortestPaths.run(g.get, Array(src))
                           .vertices
                           .filter(_._1 == config.getString("dst").toInt)

      if (r.isEmpty || r.first._2.toList.isEmpty) -1
      else r.first._2.toList.head._2
    }
  }

  override def validate(sc:SparkContext, config:Config) = SparkJobValid
}
```

然后就可以方便地进行构建：

```
sbt package
```

加载任务 Jar 包

与 Spark Job Server 进行交互，需要通过 REST 调用的方式。在这里我们会使用 curl 来进行 REST 调用，但一般情况下你可以使用 Java 或 Scala 程序来处理。

下述代码将任务 Jar 包提交给 Spark Job Server，并为它打上标签 sd。然后 Spark Job Server 会创建一个名为 sdcontext 的 SparkContext：

```
curl --data-binary @/home/cloudera/sjsslashdot/target/
➥ scala-2.10/sjsslashdot_2.10-0.1-SNAPSHOT.jar localhost:8090/jars/sd
curl -d "" 'localhost:8090/contexts/sdcontext
```

示例查询

对我们的“数据库”进行访问时，需要涉及更多的 REST 形式的查询。在 sjsslashdot.scala 文件中，我们提供了如何计算任意两个用户间的分离程度的方法。在接下来的代码中，我们对 Slashdot 的用户 0 和用户 1000 进行计算，他们的分离程度值为 2。请注意，我们使用了较多的 REST 接口参数来对 Spark Job Server 的标记位进行设置，但一般情况下不需要使用它们。首先，我们设置 sync=true 同步调用；正常情况下，在 RESTful 中对于长时间执行的函数，应当异步调用并轮询执行结果是否完成。第二，与此同时，我们指定一个 100 毫秒的稍长的超时时间：

```
curl -d '{"src":0, "dst":1000}' 'localhost:8090/jobs?appName=sd
➥ &classPath=Degrees&context=sdcontext&sync=true&timeout=100'
{
    "result": 2
}
```

和预期的结果一致，一个用户与自身的分离程度为 0。

```
curl -d '{"src":1000, "dst":1000}' 'localhost:8090/jobs?appName=sd
➥ &classPath=Degrees&context=sdcontext&sync=true&timeout=100'
{
    "result": 0
}
```

有些情况下用户间的关联链路会比较长：

```
curl -d '{"src":77182, "dst":77359}' 'localhost:8090/jobs?appName=sd
➥ &classPath=Degrees&context=sdcontext&sync=true&timeout=100'
{
    "result": 10
}
```

10.3.2 更多使用 Spark Job Server 的例子

我们可以对 Spark Job Server 进行更多的研究。例如，Spark Job Server 使用了单点故障的机制，没有做内建的容错处理。你可以尝试在两台机器上运行 Spark Job Server，并搭配使用负载均衡的策略。但你需要确保所有的任务都是无状态的。所有任务的运行状态，可以通过分布式存储的 HDFS 文件和 REST 请求中带的参数进行恢复。

Spark Job Server 提供了创建“命名 RDD”的方式，我们之前没有采用这种方法，只是将 RDD 存储在一个 Scala 的 `var` 变量中，这种方式在不需要命名 RDD 特性的时候会相对便捷。

最后，我们也没有对 `override validate()` 函数进行处理。这个方法可用于决定是否允许任务开始接受请求，从而进入处理请求的长期运行状态。

本节提及的是只读操作的示例。如果希望对图进行更新，需要提供自定义的同步 / 锁机制。但在 Spark Job Server 上使用 GraphX 的初衷，就是避免启动另一个

Neo4j 类型的计算机集群。

10.4 通过GraphFrames在Spark的图上使用SQL

对 Spark 的图处理进行展望，有一个新的图处理的库 GraphFrames，尽管它没有被包含在 Spark 1.6 版本中，但提供了很好的性能和更简单的查询方式。它提供了很多和 GraphX 类似的功能，并且还额外提供了使用 Cypher 语言（来自于 Neo4j）和 SQL 进行查询的能力。在本节中，我们会介绍一些基本的用法，测试它的性能，并将第 8 章简化版的图扩展成更复杂的形式。

GraphFrames 使用了 Spark 的 Spark SQL 部件和相关的 DataFrames 接口。DataFrames 在性能上比使用 RDD 的 GraphX 更为优越，因为它有效地使用了 Spark SQL 提供的两个最优化处理层，即 Catalyst 和 Tungsten，如图 10.3 所示。Catalyst 是 Spark SQL 在 AMPLab 时的原始名字，但现在指的是 Spark SQL 中数据库类型的查询计划优化器。Tungsten 则是 Spark 1.4 新引入的一层，用于加速内存访问的速度，它采用了 C++ 类型的绕过 JVM 直接访问内存的方式，即使用了 `sun.misc.unsafe` 的方法。

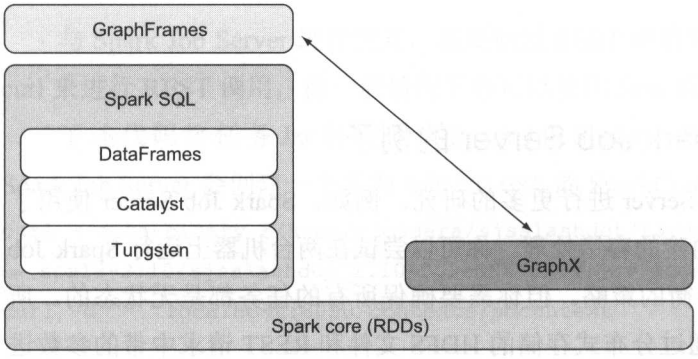


图 10.3 由于 GraphFrames 是基于 DataFrames 而不是 RDD 进行处理的，借力于 Spark SQL 内建的 Catalyst 和 Tungsten 两层的性能优化，它的处理速度比 GraphX 要快得多。Catalyst（查询优化器）和 Tungsten（绕过 JVM 的直接内存管理器），它们就像涡轮增压器插件一样极大提升了 Spark SQL 的性能。但是 GraphX 没有办法对 join() 操作进行优化，例如，一定要经过 JVM 进行所有的内存操作。

若需要更深入了解 Spark SQL，请查看 Petar Zečević 和 Marko Bonaći 编写的 *Spark in Action* 一书（Manning，2016）。对于熟悉 Python 的读者，GraphFrames 在一开始就提供了 Python 的接口，但使用 Python Spark SQL 时仍需要了解 SQL 相关的内容。

在我们使用的 GraphFrames 版本中，可使用 `AggregateMessagesBuilder` 类进行 Map/Reduce 类型的操作，这和 GraphX `aggregateMessages()` 的目标一致，但它不提供 Pregel 类型的接口。GraphFrames 的优点在于对图进行查询，而不是像 GraphX 一样用于大规模并行算法的处理，但仍需要了解在不同应用上两者运行速度的差异。GraphX 由于采用最优化的方法在内部保存了顶点和边的路由表，所以它可以快速响应三元组的处理需求。但 GraphFrames 则提供了 Catalyst 和 Tungsten 两层的性能优化处理，这是 GraphX 所没有的。

10.4.1 GraphFrames 和 GraphX 的互操作性

在 Spark 1.6 版本中，GraphFrames 并不在官方的 GitHub 目录下。在更新的版本中，GraphFrames 有机会被放在 spark-packages.org 上，或者直接作为官方 Spark 版本的组成部分。请使用以下命令来下载和构建本书所使用的 GraphFrames 版本（关于 Git 的更多操作，请查看 Mike McQuaid 编写的 *Git in Practice* 一书，由 Manning 出版社在 2014 年出版）：

```
cd ~
git clone https://github.com/graphframes/graphframes.git
cd graphframes
git checkout b9f3a30
sbt package
```

然后，使用 GraphFrames 的 Jar 包来登录 Spark REPL：

```
./spark-shell --jars ~/graphframes/target/scala-2.10/graphframes_2.10-0.0.1-
SNAPSHOT.jar
```

GraphFrames 使用 `GraphFrame` 作为基础的图类型。一个 `GraphFrame` 包含了两个 Spark SQL 的 DataFrames 类型的数据（如图 10.4 所示），这里顶点需要包含一个命名为 `id` 的数据列，边需要包含命名为 `src` 和 `dst` 的两个数据列。

GraphFrames 的 API 接口提供了与 GraphX 进行相互转换的函数。例如，假设 myGraph 是按照清单 4.1 的方法定义的：

```
import org.graphframes._
val gf = GraphFrame.fromGraphX(myGraph)
val g = gf.toGraphX
```

尽管如此，也请注意，当将 GraphFrames 转换回 GraphX 时，参数化的 VertexRDD 和 EdgeRDD 是基于 Spark SQL 的 Row 进行处理的，而不是其他用户自定义的类型安全的数据类型。

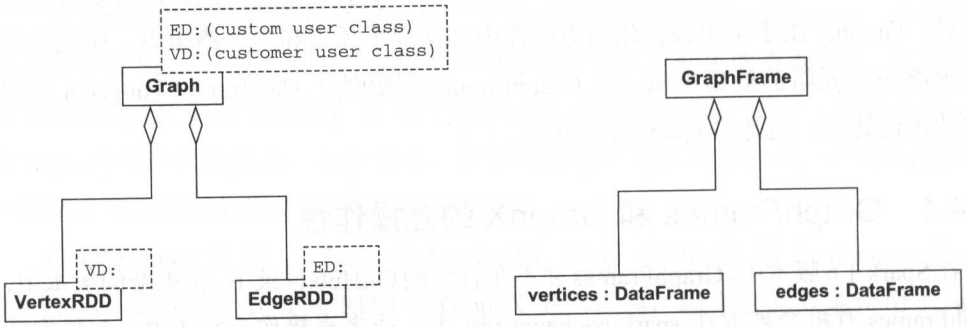


图 10.4 在 GraphX 中，基础的图类型是 Graph，而在 GraphFrames 中则为 GraphFrame。GraphFrames 没有采用参数化的类型系统进行管理，相反，它强制规定（在运行时）DataFrames 的列必须包含指定的名称。

示例：三角形个数统计

尽管一般情况下 GraphFrames 要比 GraphX 快得多，但也会有例外的情况：至少与当前 GraphFrames 版本对比，GraphX 的内建算法的速度更快。这是因为当前版本的 GraphFrames 会首先将 GraphFrame 转换为 GraphX 的 Graph，然后再执行 GraphX 的相关调用。假设图 g2 是按照清单 5.2 进行定义的，清单 10.7 中的代码展示了在 GraphFrames 和 GraphX 中进行三角形个数统计时的性能差异。

清单 10.7 GraphFrames 中三角形个数统计的标杆

```
import org.graphframes._
val gf = GraphFrame.fromGraphX(g2)
def time[A](f: => A) = {
  val s = System.nanoTime
  val ret = f
  println("time: " + (System.nanoTime-s)/1e9 + "sec")
  ret
}

time { g2.triangleCount.vertices.map(_._2).reduce(_ + _) }
time: 3.562754321sec
```

```
    res0: Int = 2592813

time { gf.triangleCount.run.vertices.groupBy().sum("count")
       .collect()(0)(0).asInstanceOf[Long] }
time: 6.493085995sec

res1: Long = 2592813
```

这段代码使用了 DataFrame 的函数 groupBy() 和 sum() 来完成聚集过程，但在下一小节中我们将会介绍如何使用 SQL 完成这一操作。

10.4.2 使用 SQL 进行便捷、高性能的操作

在本小节中，我们将会看到 GraphFrames 函数的一些特性，不仅因为使用 SQL 实现起来更方便，而且执行时还会有更高的效率。在 8.5 节中，我们展示了如何读取 RDF 文件，一种用于存储图的“三元组”的标准文件格式。从 GraphX 的角度而言（同样也是 GraphFrames 要面对的问题），上述操作的挑战在于如何为顶点分配 ID，以及如何将顶点的名称与自定义的顶点 ID 进行匹配。这是由于 RDF 文件没有顶点 ID 的数据，只有顶点的标签。为了完成这个步骤，清单 8.5 中的 readRdf() 函数包含了大量复杂的 join() 和重匹配操作。当基于 GraphFrames 实现时，不仅代码更简单和容易，性能也提高了 8 倍，代码如清单 10.8 所示。

清单 10.8 基于 GraphFrames 重写 readRdf() 函数

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
def readRdfDf(sc:org.apache.spark.SparkContext, filename:String) = {
  val r = sc.textFile(filename).map(_.split("\t"))
  val v = r.map(_(1)).union(r.map(_(3))).distinct.zipWithIndex.map(
                  x => Row(x._2,x._1))
  // 必须在顶点 DataFrame 中有个“id”列；其他的是赋予顶点的属性列表
  val stv = StructType(StructField("id",LongType) ::
                       StructField("attr",StringType) :: Nil)
  val sqlContext = new org.apache.spark.sql.SQLContext(sc)
  val vdf = sqlContext.createDataFrame(v,stv)
  vdf.registerTempTable("v")
  val str = StructType(StructField("rdfId",StringType) ::
```

```
                    StructField("subject",StringType) ::
                    StructField("predicate",StringType) ::
                    StructField("object",StringType) :: Nil)
  sqlContext.createDataFrame(r.map(Row.fromSeq(_)),str)
          .registerTempTable("r")
  // 必须在边 DataFrame 中有个“src”列和“dst”列；其他的是赋予边的属性列表
  val edf = sqlContext.sql("SELECT vsubject.id AS src," +
                           "       vobject.id AS dst," +
                           "       predicate AS attr " +
                           "FROM   r " +
                           "JOIN   v AS vsubject" +
                           "  ON   subject=vsubject.attr " +
                           "JOIN   v AS vobject" +
                           "  ON   object=vobject.attr")
  GraphFrame(vdf,edf)
}
```

新的代码实现依旧会使用到 RDD。这是因为在 Spark 1.6 版本中，DataFrames 不存在 `zip()` 或 `zipWithIndex()` 的操作，而来回地在 RDD 和 `DataFrame` 间进行转换会导致更差的性能。Jira SPARK-7460 就提出要给 DataFrame 增加 `zip()` 操作。

10.4.3 使用 Cypher 语言的子集来进行顶点搜索

在 3.3.4 节中，我们看到，对于问题“搜索 Ann 的朋友的朋友”（参考图 10.5），使用基于 Neo4j 图数据库技术的 Cypher 查询语言进行处理时相当容易。在 GraphX 中完成这样的查询时，需要编写较多行的复杂代码。而使用如 Cypher 等查询语言时，只需要一两行简单的查询代码。

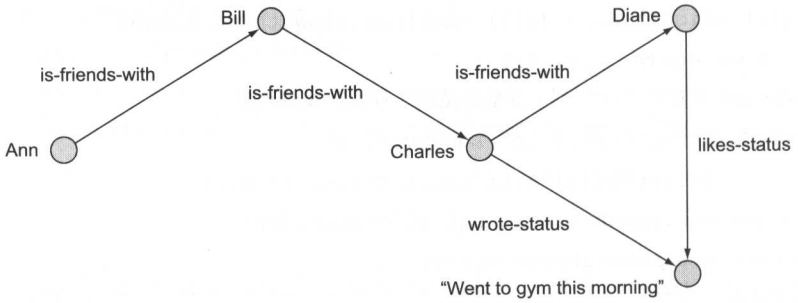


图 10.5 清单 4.1 对应的 myGraph 示例，在这里再次进行展示。

对清单 4.1 的 myGraph 进行查询

GraphFrames 支持了一个有限的 Cypher 子集，用于简化查询操作。在这个所支持的 Cypher 子集中，GraphFrames 不允许进行顶点或边的名称的直接匹配操作。它只支持对小规模的图结构进行查询。而对顶点或边的名称进行查询，则需要在下一步中基于标准的 Spark SQL 查询方法进行实现。

接下来，假设清单 4.1 中的 myGraph 已经被 Spark REPL 加载，清单 10.9 中的代码会找到 Ann 的朋友的朋友。

清单 10.9 使用 Cypher 语言子集查询 Ann 的朋友的朋友

```
val gf = GraphFrame.fromGraphX(myGraph)
gf.find("(u)-[e1]->(v); (v)-[e2]->(w)")
  .filter("e1.attr = 'is-friends-with' AND " +
          "e2.attr = 'is-friends-with' AND " +
          "u.attr='Ann'")
  .select("w.attr")
  .collect
  .map(_(0).toString)
res2: Array[String] = Array(Charles)
```

语法 `()-[]->()` 的目的是调用图表，这里 `()` 代表了对应的顶点，而边的标签则被包含在 `[]` 内。将变量占位符放在 `()` 或 `[]` 中是可选的操作，但如果采用了这样的操作，则可以在随后的 Spark SQL 查询中对它进行查询。同时，在 `()` 中重复出现的变量占位符名称代表了同一顶点，会影响相应的图结构。示例详见图 10.6。

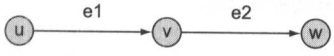


图 10.6 使用 Cypher 语法 (u)-[e1]->(v); (v)-[e2]->w 得到的子图示例。这会找到所有匹配这一结构的子图——即，第一条边的终止点和第二条边的起始点是同一顶点。

这一段代码比在 GraphX 中使用 `aggregateMessages()` 的方法更为简单。这是由于 `aggregateMessages()` 的设计初衷是为了处理大规模并行图计算，而不是查询特定的图顶点和子图结构。

GraphFrames 和 GraphX 两者的 triplets() 函数的差异

GraphFrames 也提供了 `triplets()` 函数，但正如你所想的那样，它返回的是 DataFrame 而不是 RDD。GraphFrames.triplets 的实现方法特别简单。除去字符串常

量部分，假设 gf 是 GraphFrame 类型的，下面的 Cypher 代码看起来相当美观：

```
gf.find("(src)-[edge]->(dst)")
```

现在，如果你在 Spark REPL 中对上一小节的 gf 变量（来自于 myGraph）执行 triplets() 操作，那么所得结果看起来会有一些复杂：

```
scala> gf.triplets.show
+--------------------+-----------+--------------------+

|                edge|        src|                 dst|
+--------------------+-----------+--------------------+
|[1,2,is-friends-w...|    [1,Ann]|             [2,Bill]|
|[2,3,is-friends-w...|   [2,Bill]|          [3,Charles]|
|[3,4,is-friends-w...|[3,Charles]|            [4,Diane]|
|  [3,5,Wrote-status]|[3,Charles]|[5,Went to gym th...|
|  [4,5,Likes-status]|  [4,Diane]|[5,Went to gym th...|
+--------------------+-----------+--------------------+
```

DataFrame 结果的每一列都是一个结构体（struct）。从 SQL:1999 开始引进了结构体，这意味着与 SQL 更为熟知的部分相比，我们得到的是一个“新的”特征。Spark SQL DataFrames 确实支持结构体类型，但并不总是能友好地进行处理，至少在 Spark 1.6 版本中如此。例如，为了检索边的属性，你需要知道它的名称是 attr，它是 String 类型的。你需要显式地指定名称，并显式地进行类型转换：

```
scala> gf.triplets.select("edge.attr").map(_(0).toString).collect
res3: Array[String] = Array(is-friends-with, is-friends-with,
is-friends-with, Wrote-status, Likes-status)
```

不可以将 edge 结构体的类型转换成 List 类型的属性集合，例如，转换成 Scala 的 Tuple 或 Map 类型。

10.4.4 稍微复杂一些的 YAGO 图同构搜索

在 8.3 节中，我们通过推荐算法 SVD++ 来扩充 YAGO 图的边集，从而完成了 Wikipedia 潜在缺失数据的推测。可以看到，基于从 YAGO 大图中挖掘到的国家和出口商品间的顶点对，可以推测出加拿大很有可能也出口电子设备产品。

通过使用 Cypher 语言，不仅可以查找顶点对，还可以查找三角形。即，我们可

以找到缺失了一条边的三角形，如图 10.7 所示。

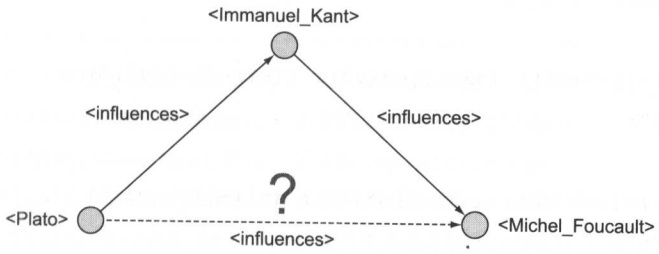


图 10.7 使用 Cypher，找到的 Wikipedia 可能缺失的边信息。

在这个例子中，我们同样使用到了 yagoFacts.tsv 文件，但考虑的是人与人相互影响的关系（哲学家、诗人、艺术家等），即标签为“<influences>”的边。我们要找到一些具有高影响力的人，如 Plato，即在统计学上来看，所有他们间接影响的人，也会被他们直接影响到。在清单 10.10 里，变量 absent（对应表的别名为 a）是 DataFrame 类型的，包含了所有缺失了第三条边的三角形。而变量 present（对应表的别名为 p）也是 DataFrame 类型的，包含了所有具有三条边的三角形。计算变量 absent 的过程中就使用到了 Cypher 的操作符 !（感叹号），用于查找缺失的边。

在执行清单 10.10 中的代码前，首先使用 grep 命令来减少 yagoFacts.tsv 文件的大小，只保留包含“<influences>”的行：

```
grep "<influences>" yagoFacts.tsv >yagoFactsInfluences.tsv
```

清单 10.10 使用 Cypher 和图同构来查找缺失的 Wikipedia 信息

```
val in = readRdfDf(sc, "yagoFactsInfluences.tsv")

in.edges.registerTempTable("e")
in.vertices.registerTempTable("v")

val in2 = GraphFrame(in.vertices.sqlContext.sql(
                    "SELECT v.id," +
                    "       FIRST(v.attr) AS attr," +
                    "       COUNT(*) AS outdegree " +
                    "FROM   v " +
                    "JOIN   e " +
                    "  ON   v.id=e.src " +
```

```
                    "GROUP  BY v.id").cache,
                  in.edges)

val absent = in2.find("(v1)-[]->(v2); (v2)-[]->(v3); !(v1)-[]->(v3)")
absent.registerTempTable("a")

val present = in2.find("(v1)-[]->(v2); (v2)-[]->(v3); (v1)-[]->(v3)")
present.registerTempTable("p")

absent.sqlContext.sql(
  "SELECT v1 an," +
  "       SUM(v1.outdegree * v2.outdegree * v3.outdegree) AS ac " +
  "FROM a " +
  "GROUP BY v1").registerTempTable("aa")

present.sqlContext.sql(
  "SELECT v1 pn," +
  "       SUM(v1.outdegree * v2.outdegree * v3.outdegree) AS pc " +
  "FROM   p " +
  "GROUP BY v1").registerTempTable("pa")

absent.sqlContext.sql("SELECT an," +
                      "       ac * pc/(ac+pc) AS score " +
                      "FROM   aa " +
                      "JOIN   pa" +
                      "  ON   an=pn " +
                      "ORDER BY score DESC").show
```

上述查询的结果如下所示，Plato 是最有影响力的人，所有被他间接影响到的人（即与他的分离程度为 2 的人群），也能被他直接影响到。在计算过程中，我们使用了一个打分公式。打分公式的关键部分是 pc/(ac+pc)，是将完全三角形的个数除以所有三角形的个数（包括完全和不完全三角形），然后再将它和 ac 相乘。这是我们为这些名人提出的启发式计算方法。

```
+--------------------+--------------------+
|                  an|               score|
+--------------------+--------------------+
|  [7662,<Plato>,102]| 3.822406412297308E7|
|[10648,<Aristotle...|3.2961326121938106E7|
|[4959,<Immanuel_K...|2.6445857520978764E7|
|[2961,<Georg_Wilh...|2.1092802441273782E7|
|[9304,<Baruch_Spi...|1.4513392385496272E7|
|[12217,<René_Desc...|1.2407118036818413E7|
|[12660,<Johann_Wo...|1.0109121178397963E7|
|[11895,<Jean-Jacq...|   9081581.748842742|
|[11615,<Gottfried...|   7146037.710399863|
|[2025,<Friedrich_...|  6897244.1896990575|
|[1082,<William_Sh...|   4168778.144288711|
|[11034,<Adam_Smit...|  4100936.5022027283|
|[1121,<John_Locke...|   3868447.819527024|
|[1566,<Heraclitus...|  3616900.3025887734|
|[3746,<Karl_Marx>...|   3575419.671920321|
|[10954,<Søren_Kie...|   3143375.914849735|
|[7322,<David_Hume...|  3122089.3473657905|
|[8540,<Arthur_Sch...|   2978239.727690162|
|[3186,<Ibn_Tufail...|   2234249.031615453|
|[8267,<Epicurus>,24]|  1812594.4073720106|
+--------------------+--------------------+
only showing top 20 rows
```

在最后的查询中，我们限制为只查 Plato 相关的信息（在之前的结果中对应 ID 为 7662 的顶点），并从“缺失”表中查找关联的人（即查找缺少第三条边的三角形）。同样的，我们会偏好三个顶点的度值均较高的三角形，从而找出所需要的名人。而使用 SQL 计算顶点度值相当简单。考虑以 Plato 为起始点的可能缺失的边，在我们得到的结果里，最高得分的两个候选者是 Marx 和 Sartre。但他们的观点与 Plato 的正好相反，这和 Wikipedia“<influences>”标签的用法不一致，所以不应该说他们受到了 Plato 的影响。列表的第三位候选者，Foucault，曾亲切地讨论过 Plato 的作品，但更多的是从分析者的立场给出了他自己的哲学观点。所以在这个例子中，是否适合应用“<influences>”边的做法值得考虑：

```
absent.sqlContext.sql(
  "SELECT v1.attr, " +
  "       v3.attr, " +
  "       SUM(v1.outdegree * v2.outdegree * v3.outdegree) AS score " +
  "FROM   a " +
            "WHERE v1.id=7662 " +
            "GROUP BY v1.attr, v3.attr " +
            "ORDER BY score DESC").collect
res24: Array[org.apache.spark.sql.Row] = Array([<Plato>,<Karl_Marx>,7139388],
[<Plato>,<Jean-Paul_Sartre>,3143640], [<Plato>,<Michel_Foucault>,2871606],
[<Plato>,<Gilles_Deleuze>,2689128], [<Plato>,<Henri_Bergson>,2179128],
     [<Plato>,<Maurice_Merleau-Ponty>,2088450]...
```

打分机制的一个可行改进是，惩罚那些出生日期差异极大的搜索结果。毕竟，若认为一个古代哲学家会直接“影响”到一个已接受现代哲学的现代哲学家，这种观点本身是否合适呢？改进后的查询会涉及另一个 YAGO 文件——yagoDateFacts.tsv，并需要将它和 yagoFacts 图进行合并，然后查询更复杂的、涉及“<wasBornOnDate>”类型的边的子图。

10.5 小结

- Scala 是 Spark 和 GraphX 的原生语言。
- 使用 Java 7 处理 GraphX 会很复杂，与 Scala 相比需要 10 倍的代码量。
- 使用 Java 8 处理 GraphX 的代码量仅比使用 Java 7 时稍微少一点，这是因为 Java 8 的 lambda 表达式仅适用于部分情况。
- 在 Spark 1.6 中，Spark 支持 R 和 Python 语言，但是否以及何时会被 GraphX 支持仍不确定。
- 组合使用 Zeppelin 和 d3.js，可以提供和 REPL 匹敌的强大的交互式能力，并可提供内联的图可视化能力。
- 需要对图可视化做更深入的操作时，需要相关的 d3.js 的知识，或使用 Gephi 进行处理。
- Spark Job Server 为 Spark 增加了 REST 形式的接口，这意味着使用 GraphX 时，图数据可以保留在 RDD 中，并提供类似数据库的处理形式。

- GraphFrames 是一个新的库，它通过使用 SQL 和 Cypher 语言的子集，可以更方便地对特定顶点或子图进行查询。
- GraphFrames，由于是基于 Spark SQL 建立的，在 Spark SQL 内建的 Catalyst 和 Tungsten 这两个优化层的帮助下性能很卓越。

附录A　安装Spark

本附录要点

- Spark 快速入门
- 使用虚拟机运行
- 使用 AWS 的弹性 Map/Reduce（EMR）运行 Spark

开始使用 Spark 之前，首先要有一个集群，其次在这个集群中的每台机器上安装 Hadoop。如果你正在做 GraphX 相关的工作或者工作中有可用的 Hadoop/Spark 集群，可以尝试在这些集群上练习使用。如果没有可用的集群，下面会有几种不需要 Hadoop 或集群机器的选择。

下面给出了三个选择。

1. 本地虚拟机模式：Cloudera QuickStart VM（预安装了 Hadoop 和 Spark）。
2. 在 Linux、OS X 系统的笔记本、台式机（虚拟机上：Hadoop 非必需）。
3. 云主机：亚马逊 Web 服务（AWS）。

一些开发人员更喜欢在虚拟机上开发，本附录也提供了这种不常见的做法（在这里，我们指的是笔记本电脑中的虚拟机 VMware Player 或 VirtualBox，不是云上的虚拟机）。多个虚拟机允许开发人员容易地在多个项目中工作，每个项目都有自己的环境，不同的 Java 版本、Scala 版本、操作系统版本等。并且虚拟机很容易移交给同事和团队成员。最后一个好处，开发人员可以在虚拟机和宿主机之间复制粘贴电子邮件、常用的工具和数据文件。

虽然 Spark 可以使用 Cygwin，但也极不推荐直接在 Windows 中使用 Spark。在 Mac OS X 上要好得多，这是由于 OS X 也是出自 BSD UNIX。不过，我们推荐在装有 CentOS 的虚拟机上运行 Spark，CentOS 是企业版红帽 Linux 的社区版，在 2014 年得到红帽的支持。红帽 Linux、如 CentOS 这类 Linux 派生版本、Oracle Linux 这些 Linux 系统在企业中广泛使用，所以在 CentOS 环境下开发 Spark 可以确保与生产环境相近。

A.1 本地虚拟机模式：快速入门级的虚拟机

最方便的方式是下载 Cloudera QuickStart VM，可以在三种虚拟机上使用 CDH 的软件包：VMWare、VirtualBox 和 KVM。这三类虚拟机各有不同的特性和不同的许可证。VirtualBox 被用得最多，因为它是通用性公开许可证（General Public License，简称 GPL），可以免费使用，也可用于商业用途。这里没有用 KVM，因为 KVM 要求 Linux 作为宿主系统，我们常用的系统是 Windows 或 OS X。

Cloudera QuickStart VM 中预安装了 Hadoop 包（称为 CDH），CDH 中也包含了 Spark 和 GraphX。

使用 Cloudera 虚拟机前有以下要求：

- 至少 8GB 的物理内存，因为虚拟机需要 4GB 内存。
- 网络够快（或者足够的耐心），要有 3GB 的 .7z 压缩文件要下载。
- 配置 BIOS 以允许运行 64 位的虚拟机。

上述最后一项比较棘手。出于安全原因，大多数计算机都默认禁用主机允许运行 64 位的访客虚拟机，即使它们是 64 位的电脑能够运行 64 位操作系统。首先，如果你不知道如何进入你的电脑的 BIOS 设置，利用搜索引擎查询一下。其次，一旦进入 BIOS 设置界面还找不到允许运行 64 位访客虚拟机的设置，可能需要在搜索引擎中搜索与你的计算机型号名称相关的关键字“BIOS VT-x”（英特尔处理器）或“BIOS AMD-V”（AMD 处理器）。

启动 Cloudera QuickStart VM，用下面的账号登录：

Username：cloudera

Password：cloudera

Cloudera QuickStart VM 不好的地方是，它默认用 4GB 内存，虚拟机启动后会自动启动 Hadoop 服务，只剩下 800MB 内存可用。Spark 中一般是不变的数据，特别是 GraphX，这就限制了实际最大可用数据集为 30MB 左右。另外一个不足之处是，相比于 AWS 或物理机集群，Cloudera QuickStart VM 会限制并行，这可是 Spark 的主要优点。Spark 只能在多核的单块 CPU 上并行计算，这样就无法让 Spark 在运算大数据集时性能有极大的提升。

A.1.1 VirtualBox 微调

如果你选择了 VirtualBox 版的 Cloudera QuickStart VM，下面是一些小窍门和微调：

- 下载并解压后，使用“导入”而不是“新建”或“打开”来初始化加载虚拟机，这对熟悉 VMware 的人来说会比较困惑。
- 在 VirtualBox 的管理窗口中设定虚拟机的各种配置设置。
- 快速启动虚拟机会默认一个 CPU 核，由于现在计算机大都是 4 核、8 核甚至更多核，通过 System > Processor > Processors 来增加多核。
- 如果宿主机内存超过 8GB，那就给虚拟机多分配一些内存，通过 System > Motherboard > Base Memory 进行设置。一个最佳实践是保留 4GB，其余分配给虚拟机用（例如，有 16GB 物理内存，那就将 12GB 给虚拟机用）。
- 复制粘贴操作在虚拟机和宿主机之间非常有用，默认是禁用的。通过 General > Advanced > Shared Clipboard 可修改“双向设置”。
- 文件共享设置。要添加一个共享目录，单击文件夹图标用加号创建一个共享目录。例如，在 Windows 下，“C:”盘下有小写的“c”目录，启动虚拟机后，在命令行窗口中输入：

```
sudo mkdir /c
sudo mount -t vboxsf c /c
```

A.2 在不需要Hadoop的笔记本操作系统上：Linux或OS X

事实上，开发环境中运行 Spark 真的不需要 Hadoop。如果集群中只有一个节点，确实不需要 Hadoop，这样 Spark 中唯一用到 Hadoop 的地方是读写文件，如果

Spark 只需要访问本地文件系统，更没有必要用 HDFS、Cassandra 或亚马逊 S3 这类分布式存储系统了。

简言之，如果目标只是熟悉 GraphX 的 API，不需要操作超大数据集及性能调优，在一个类 UNIX 的操作系统上（Linux 或 OS X）只安装 Spark 是完全可行的。

只安装 Spark 适用于以下情形：

- 笔记本或台式机中安装了 Linux 操作系统。
- 笔记本或台式机是双系统启动（例如，使用 Grub 或 BootIt Bare Metal），其中一个是 Linux 系统。
- 笔记本或台式机是 OS X 系统。
- 创建一个虚拟机，其操作系统是 Linux。
- 使用云虚拟机，如亚马逊的 AWS、Azure 或者托管的虚拟机。

你可以从 Apache 网站下载各种已打包好的 Hadoop 版本（Hadoop 1.x，Hadoop 2.x，MapR 等）和预构建打包的 Spark 版本。在这种情况中，我们根本不使用 Hadoop，你选择哪一个都没有关系，只要不尝试读或写 HDFS 文件。

下载 Spark tgz 文件，将其解压缩，准备开始。要用到 Spark Shell，甚至不需要安装 Scala，只需要 Java。但是，要是构建 Scala 写的 Spark 程序，那就需要安装 Scala 了。

A.2.1 定制的本地虚拟机

结合上面两种 Spark 环境搭建方式，无 Hadoop 加虚拟机是比较好的选择，它们有以下优点：

- 相较于预构建的 Cloudera QuickStart VM，不启动 HDFS 和其他 Hadoop 服务，会节省 1 ～ 2GB 内存。
- 相较于在操作系统中安装 Spark，虚拟机有本附录中提到的好处。

从零创建一个虚拟机，搭建环境，并不是容易的事情。需要配置正确，调整许多地方，这里不一一赘述，读者可以自行搜索，网络中有很多有用的帮助文档。或者使用预先构建好的虚拟机，其中包括所需软件及 Linux 系统。

A.3 在云上：AWS（Amazon Web Services）

AWS 提供了几十个不同的云服务，最有名的是用于存储的 S3 和用于弹性计算的 EC2，弹性 MapReduce（EMR）提供了 Hadoop 和 Spark 的计算服务。EMR 通过 S3 和 EC2 资源来支撑整个 Hadoop 集群（Spark 可有可无）。

相比于前面提到的几个集群部署方式，AWS EMR 的优势在于用户可以运行一个真正的集群，实现并行计算，处理大数据集，开发的 Spark 应用程序可以运行在 YARN 集群上。

但是 AWS 不是免费的。另一个问题是，如果使用自动启动的 Spark 集群，不能暂停。要暂停它，意味着必须彻底释放集群，到目前为止还没暂停集群而不收费的功能。这意味着一旦启动集群就要持续利用集群运算出结果，保存在 S3 上。没有办法暂停 AWS Spark 集群的同时暂停计费，这意味着你必须认真保存你的计算结果到 S3。也没有办法把数据存储在 REPL 中，过些时间回来继续使用。

附录B　Gephi可视化软件

第 4 章中有代码可以生成以 .gexf 为后缀的 Gephi 文件。下载和安装 Gephi 很简单（它可用于 Windows、OS X 和 Linux），下载地址为 https://gephi.org/users/download/，但它的用户界面有些特殊。本附录会介绍 Gephi 最重要的用户界面元素，足以让你入门，然后你自己就可以探索其余丰富的功能了。

B.1　环境布局

Gephi 有可停靠的窗口，更像是个 IDE。从图 B.1 中可以看到，如何用 Gephi 生成本书所需的图。下图中的三个停靠窗口（Overview、Data Laboratory、Preview）可以从图 B.2 所示的下拉菜单中找到。一旦打开这些窗口，就可以如图 B.1 一样在编辑区拖曳图元素。

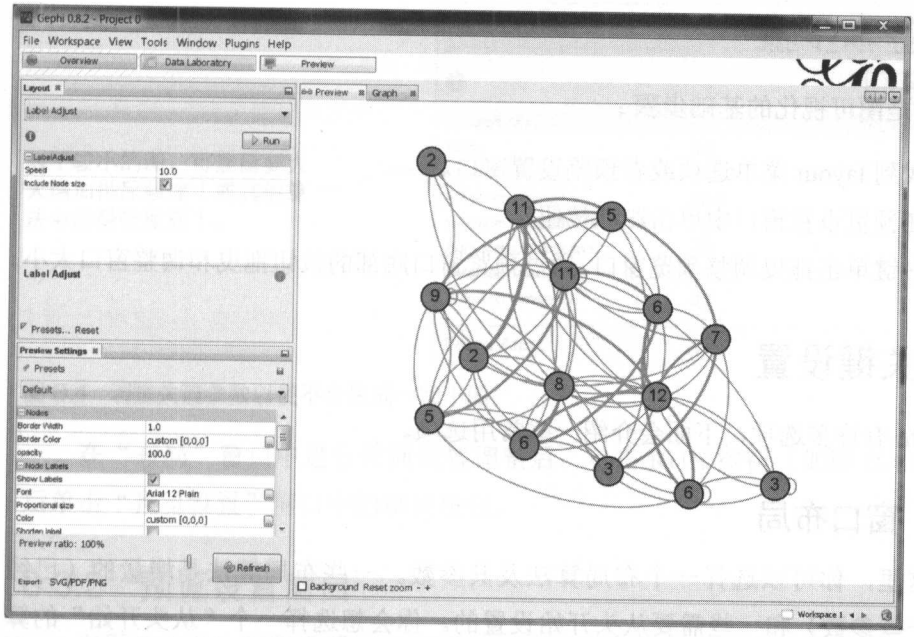


图 B.1　Gephi 的窗口布局

注意看 Graph 选项卡，从位置和名称来看像一个重要的选项卡，但它是在图上进行处理。我们通常在 GraphX 中进行图处理，只是对 Gephi 的可视化功能感兴趣，一般会忽略 Graph 选项卡。

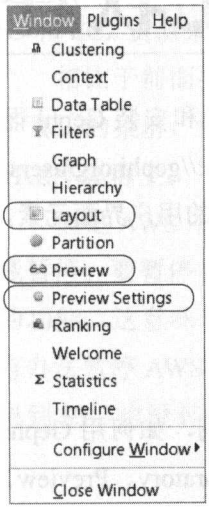


图 B.2 从下拉菜单中选择打开 3 个窗口。

B.2 基本步骤

以下是图可视化的基础步骤：

1. 找到 layout 菜单选项或者预览设置窗口。
2. 在预览设置窗口中单击刷新按钮。
3. 右键单击拖曳调整预览窗口，通过预览窗口底部的按钮拖曳和调整窗口大小。

B.3 关键设置

Gephi 有许多选项，下面会介绍一些常用选项。

B.3.1 窗口布局

在这里，你可以选择一个布局算法及其参数。一些布局算法是增量的（已经预设了一些参数）和一些需要从头开始设置的。你会想选择一个“从头开始”的算法，然后在必要时调整成一个“增量”算法。现有的算法如图 B.3 所示。通常 Force

Atlas 是一个很好的起点，因为它能可靠地产生合理的结果。

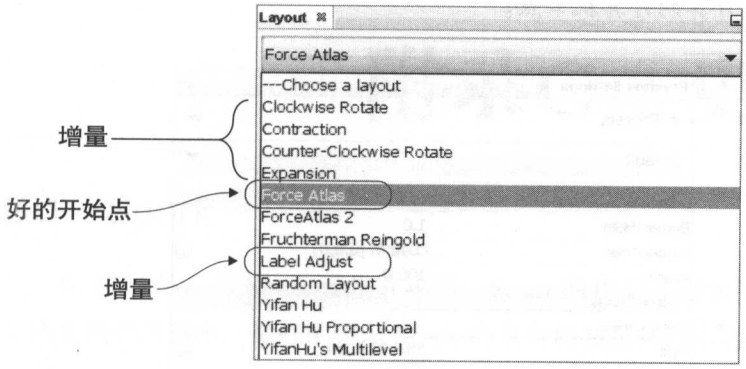


图 B.3 在布局窗口下拉列表中的可用布局算法。那些没有标记为增量的都是头等布局算法，其从零开始执行一个完整的布局。增量的用在一个已经布局的图上。

对于小图，你可能需要首先将“排斥强度”（或其他算法中的“最佳距离”）调整为更大的数，如图 B.4 所示。Gephi 被设计为处理具有数百或数千个顶点的非常大的图，并且其默认设置提供非常短的边。对于有十几个或几十个顶点的图，需要通过增加“排斥强度”或“最佳距离”来使边更长。

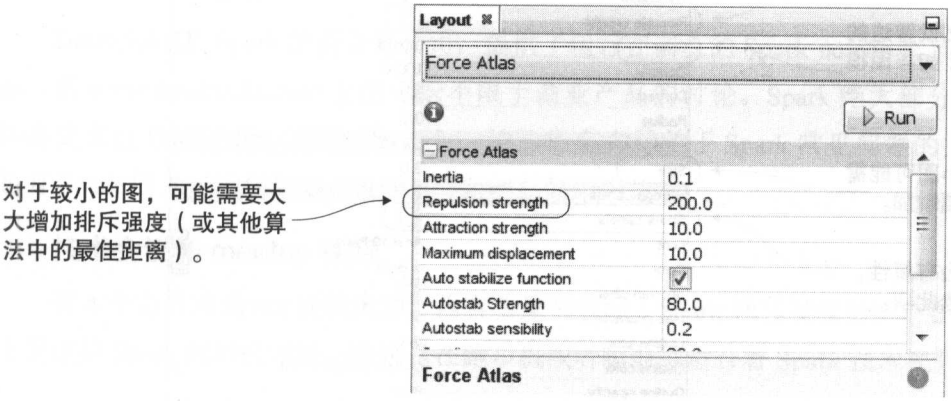


图 B.4 调整关键参数以便不会生成一团小图。

在“布局”窗口中进行任何设置调整后，单击 RUN 按钮（如图 B.4 所示），然后单击“预览设置”窗口中的刷新按钮。

B.3.2 预览设置窗口

在图 B.5 中，预览设置窗口中的一些重要的设置项已高亮显示。

注意：Gephi 中的术语“节点”是顶点的意思。本书约定“节点”为集群中参与分布式计算的计算机节点。

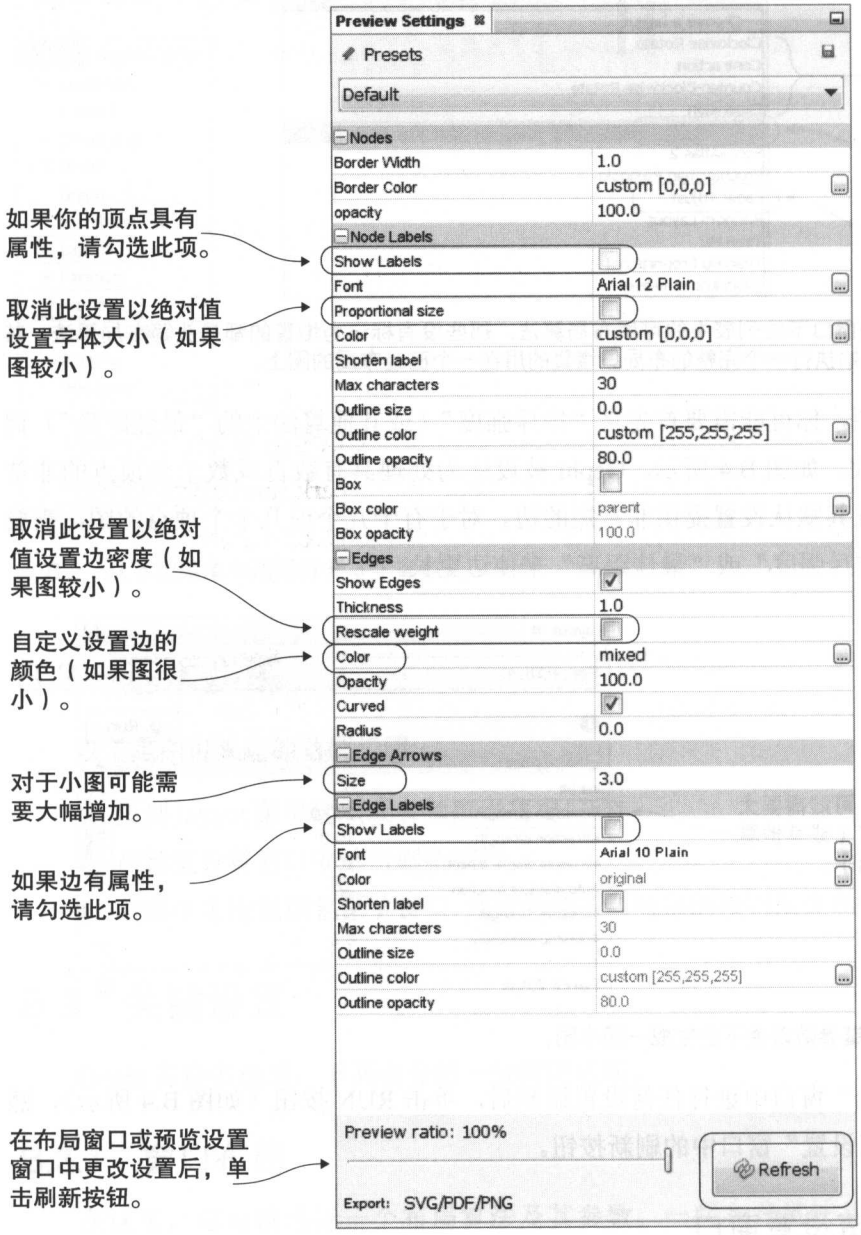


图 B.5 预览窗口的关键设置。

附录C 更多资源

C.1 Spark

在2015年，即Spark诞生6年后，Spark的相关书籍快速增多。但Spark的开发还在快速演进中，最好的学习Spark的资料仍然是网络资料。

Apache邮件列表

正如其他开源项目一样，特别是Apache的开源项目，邮件列表是最好的交流和学习的地方。遇到问题，要是在网上找不到办法，可以考虑向邮件列表中的地址发邮件进行求助，这会比较有效。邮件列表为：user@spark.apache.org、dev@spark.apache.org，从这里订阅邮件列表：https://spark.apache.org/community.html。

Databricks论坛

Databricks是Spark的商业化公司，提供了运行在云上的Spark notebook商业产品。而www.databricks.com上的论坛不限于商业产品的讨论。Spark绝大部分的代码提交来自Databricks，所以Databricks论坛也有大量关于Spark常见问题的探讨，也有Spark的未来计划和Databricks商业产品的未来计划。

技术会议和meetup视频

有4个主要的Spark视频来源，内容质量都很好。Spark社区发展很快，在手机上看这些Spark视频很方便，特别是在跑步机或者睡前很适合看Spark技术视频：

1. Spark峰会（西部、东部和欧洲峰会）
2. 美国UC伯克利大学AMP（Algorithms、Machines、People）大数据开发训练营
3. 湾区的Spark Meetup
4. O'Reilly Strata大会（西部和东部）

Jira

如果 Spark 在你当前的工作或学习中仍然很需要，最好多关注和跟随 Spark Jira。要是还没有 Apache Jira 账号，先新建一个账号，按时间倒序排列出每天所有的 Spark 话题，单击并关注（Watch）对你重要的话题。这样就可以及时了解到最新的特性、修复的 bug、性能提升、架构更迭、第三方的支持（如文件系统、集群管理、数据库连接、压缩格式、序列化方案等），更重要的是，可以了解社区最新的目标版本。

当你第一次列出 Spark Jira，会发现也有一些早期的仍然在计划进行的特性，这些特性听起来都很难弄也比较耗时，但这些仍然在进行的 Jira 或许很值得你花时间浏览一下。

Twitter

要是你觉得 Twitter 的 140 字的信息量太少而仅仅是看看明星，那么下面的介绍会让你大吃一惊的。

在 Twitter 上有很多大数据、数据科学和机器学习方面的研究。你可以用 Twitter 聚合一些热门和重要的博客、新闻或者 Git 仓库地址。

spark-packages.org

Spark 社区不想让 Spark 官方发布版本中包含过多非关键特性和附属包，所以就建了 spark-packages.org 这个网站。可以在机器学习、图、Python 等分类中添加自己的包。

AMP Lab

Spark 是从 AMP Lab 孵化并独立出来的，并且 AMP Lab 也持续开发一些与 Spark 适配的项目。这些孵化项目一般倾向于直接合并到 Apache Spark 的发布版本中（如 GraphX、Catalyst 即后来的 Spark SQL、SparkR），或者至少是半官方支持，如 Tachyon（已更名为 Alluxio）。

Google 学术提醒

大家对 Google Alerts 都很熟悉，当你监控的一个页面更新后，Google Alerts 会发邮件给你。而 scholar.google.com 中的 Google 学术提醒则完全不同，当你正跟踪的一篇论文被一个新发表的论文引用时，Google 学术提醒会给你发邮件提醒。

如果对一些重要的 Spark 论文设置了 Google 学术提醒，那么像 Matei Zaharia 的

论文 *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing* 以及 Joseph E. Gonzalez 的论文 *GraphX: Graph Processing in a Distributed Dataflow Framework*，就可以在他们开始商业化之前保持跟进其最新学术进展。

作者的博客

如果已经采纳了上述建议，其实没有必要学习下面的博客。但是你要想节省些时间概要地了解 Spark 未来版本的特性、大数据、数据科学和机器学习，可通过 Michael Malak 的个人经验来了解这些内容，以下是他的博客链接：

- http://technicaltidbit.com
- http://datascienceassn.org/blogs/michaelmalak

C.2 Scala

学习 Scala 的最好的资料是看书，但有些 Scala 书很厚，这是因为 Scala 有这么多的特性和技巧。可以选择下面的百科全书类的书籍：

- *Scala Cookbook by Alvin Alexander*（O'Reilly，2013）
- *Scala Puzzlers by Andrew Phillips*（Artima，2014）

C.3 Graphs

有许多关于图论的书，大部分是理论性的，一般作为大学教科书使用或者给研究人员使用。而下面的书或许对开发人员有用：

- *Graph-Based Natural Language Processing and Information Retrieval by Rada Mihalcea and Dragomir Radev*（Cambridge University Press，2011）
- *Graph Databases by Ian Robinson et al*（O'Reilly，2015）

附录D　本书中的Scala小贴士

本书的目的不是学习 Scala，而且可能 Scala 不是你最熟悉的编程语言，可能还没有见过所有常用的 Scala 技巧，那么可以通过小贴士来了解 Scala 的用法。Scala 的相关书籍，见附录 C。

下面是散落在本书中的 Scala 小贴士。

第 2 章

第 4 章

第 5 章

第 6 章

第 7 章

第 8 章

Broadview® www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Spark GraphX实战

GraphX是Apache Spark的一个功能强大的图处理API，用于分析大型数据集。GraphX为你提供了前所未有的速度和能力，可用于运行大规模并行图算法和机器学习算法。

本书一开始描绘了图计算的应用场景，提供了一些基础示例教你如何交互式地使用GraphX。本书还清晰地介绍了如何从常规数据构建大图，然后研究了一些问题及如何用图算法和图架构解决这些问题。通过阅读本书，你将了解到用于增强应用程序的实用技术及如何将机器学习算法应用于图数据。

本书包括：

◎了解图技术
◎使用GraphX API
◎为大图开发图算法
◎用于图的机器学习算法
◎图的可视化

通过阅读本书，即使读者没有Spark和Scala基础也可以很自然地写出应用代码。

Michael Malak从2013年就开始在财富500强公司从事Spark应用开发。**Robin East**在大型组织担任顾问超过15年，是Worldpay的数据科学家。

读者可以访问www.manning.com/books/spark-graphx-in-action下载PDF、ePub和Kindle格式的免费电子书。

“跟随两位经验丰富的作者学习复杂的图处理方法，本书是一个全面指南。”

——Gaurav Bhardwaj，3Pillar Global

“在最短的时间内，从GraphX新手到专家的最佳学习资源。”

——Justin Fister，PaperRater

“了解大规模图数据挖掘的必读书籍！”

——Antonio Magnaghi OpenMail

“揭示了处理有连接关系的大数据集的卓越且优雅的能力。”

——Sumit Pal独立顾问

MANNING

策划编辑：张春雨
责任编辑：刘 舫
封面设计：李 玲

上架建议：大数据

ISBN 978-7-121-31043-0

9 787121 310430

定价：79.00元